Vérification automatique de systèmes concurrents

École Jeunes Chercheurs en Algorithmique et Calcul Formel,
Marne-la-Vallée, 31 mars - 4 avril 2003

Anca Muscholl et Marc Zeitoun

LIAFA, Univ. Paris 7 & CNRS

Table des matières

1. Introduction

Ces dernières années, la vérification des programmes assistée par ordinateur a connu un fort développement, tant en ce qui concerne sa fondation théorique que ses applications. Son objectif est de vérifier des propriétés de validité, de sûreté,... de façon algorithmique, sans avoir besoin d'une compréhension approfondie du programme à vérifier. La méthode employée s'appuie sur les concepts de base de la théorie du calcul : les automates et la théorie des langages formels, de mots ou d'arbres. Le fondement de cette méthode est l'équivalence entre la notion algébrique de reconnaissabilité, la définissabilité logique et la reconnaissabilité par automates. Les deux premiers formalismes permettent de spécifier des propriétés des systèmes, et le troisième est utilisé dans les algorithmes de vérification. La vérification est ainsi ramenée à l'algorithmique des automates.

Pour des introductions au domaine, on peut se reporter à [7, 10].

1.1 Que signifie *vérification* dans ce cours ?

– But : vérifier de façon *automatique* (algorithmique) qu'un programme est correct.
– Théorème de Rice : toute propriété non triviale des langages récursivement énumérables est indécidable $\Longrightarrow$ hypothèses raisonnables pour rendre ce but atteignable.
– Dans ce cours : vérification de programmes réactifs concurrents.
  - Programme réactif : dans un modèle classique, un programme prend une entrée, effectue un calcul sur celle-ci, et fournit une sortie à l'issue de ce calcul. Un programme réactif, au contraire, est destiné à fonctionner en permanence et réagit à des entrées continues de la part de l'environnement dans lequel il se trouve. Exemples de programmes réactifs : les systèmes d'exploitation, les ordonnanceurs, les serveurs...
  - Programme concurrent : Programme composé de plusieurs processus (ou *threads*) qui exécutent des actions pouvant ou non influencer le comportement d'autres programmes. Par exemple, dans le cadre classique de l'exclusion mutuelle, un processus entrant dans une section critique (parce qu'il demande une ressource critique) bloque les autres processus. Par contre, des processus en section restante peuvent évoluer en parallèle.

1.2 Aperçu des problèmes abordés

1.2.1 Objectifs et difficultés de la vérification

But Validation de protocoles, d'algorithmes. L'objectif est de détecter les erreurs possibles tôt dans la conception d'un logiciel. On ne s'intéressera pas à la modélisation : le modèle sera supposé déjà donné (modéliser un système déjà existant est un travail difficile ; on peut penser à faire une modélisation *a priori* d'un système à construire, dans une phase de design qui abstrait de nombreux détails).

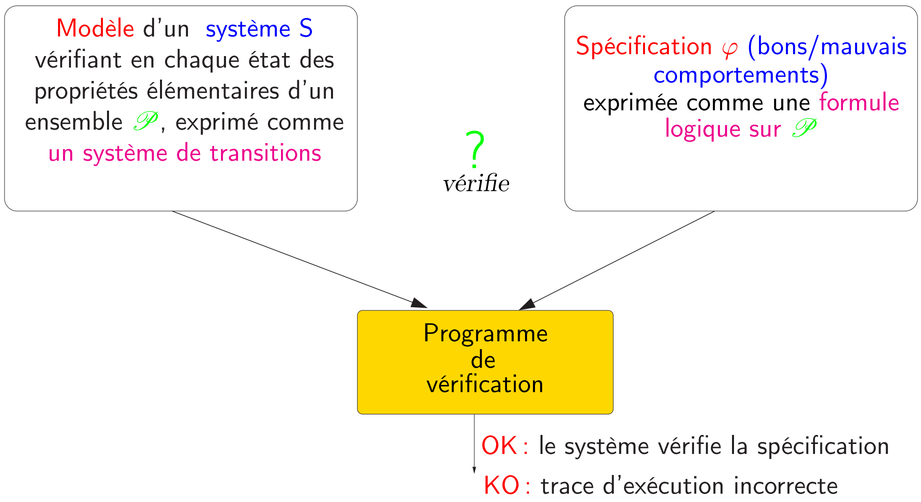


FIG. 1.1 – *Le* model-checking*, ou vérification de modèle*

Difficulté Systèmes à grands nombre d'états ou à nombre d'états non borné. Un système concurrent peuvent avoir un nombre d'états très élevé, même si chaque composant du système est petit.

Méthodes classiques fournissant des indications sur la correction d'un système

- Simulation/test :
  - l'effectivité se perd lorsque la précision du modèle s'accroît.
  - problème de complétion de la procédure (exploration de certains comportements du système, basée sur des heuristiques, recherche de couverture « adéquate »)
- Theorem proving : pas entièrement automatique (Rice), demande du temps et de l'expertise.
- Vérification formelle vérification exhaustive : tous les comportements réputés corrects.
  - Le *model-checking* est complètement automatique.
  - Pas de comportement « oublié ». Si le *model-checker* valide un système, ce système est correct vis-à-vis de la propriété vérifiée.
  - Lorsque les propriétés requises ne sont pas satisfaites, les logiciels de *model-checking* exhibent une trace du système non satisfaisante.
  - Les propriétés sont exprimées dans des logiques temporelles (Clarke/Emerson et Quielle/Sifakis), qui permettent d'exprimer *facilement* de nombreuses propriétés naturelles.
  - La méthode se généralise à des systèmes concurrents.

  L'explosion du nombre d'états à parcourir pour vérifier rend utiles des techniques de réduction de l'espace d'états.
  - techniques symboliques : ne pas représenter explicitement chaque état, mais calculer sur des représentations compactes.

– techniques réductions par ordre partiel : identifier des chemins qu'il suffit d'explorer pour garantir la vérification complète de la propriété.

1.2.2 La vérification des systèmes séquentiels

- Modélisation des comportements d'un système par un mot (qui peut être infini), sur un alphabet $2^{\mathscr{P}}$. Chaque lettre représente un ensemble de propriétés valides.
- Générateurs de comportements séquentiels : systèmes de transitions dont chaque état est étiqueté par l'ensemble de $2^{\mathscr{P}}$ des propriétés valides dans cet état.
- Spécifier des propriétés : étude de différentes logiques.
  1) Logique Temporelle Linéaire LTL : permet de parler des propriétés d'un mot sur $2^{\mathscr{P}}$. Un système de transitions vérifie une formule de cette logique si *tous* ses comportements la vérifient.
  2) Logiques arborescentes CTL et CTL*, spécifiant des propriétés directement sur le système de transitions.
- Pour chacune de ces logiques, on étudie, du point de vue algorithmique et de celui de la complexité :
  a) le problème du model-checking : étant donnés
     – un système S de transitions dont les états sont étiquetés par des propriétés élémentaires,
     – une propriété φ de LTL, CTL, CTL*...

     on veut savoir si $S \models \varphi$.
  b) le problème de la satisfaisabilité : étant donnée $\varphi(\in$ LTL,CTL$,\ldots)$, on veut savoir s'il existe un modèle qui vérifie φ. Ce problème est intéressant car il a des liens avec le précédent.
  c) le calcul de l'expressivité de la logique : quelles propriétés φ peut-elle exprimer ?

1.2.3 Les modèles de la concurrence

Souvent, les systèmes réels sont obtenus par la mise en œuvre de plusieurs composants fonctionnant ensemble. On peut bien sûr étudier le système global, fonctionnant comme un seul processus. Le problème auquel on se heurte est l'explosion du nombre d'états du système (le nombre d'états de $p_1 \| p_2$ peut-être $n_1 \times n_2$, où n_i est le nombre d'états de p_i). On peut également essayer de tirer parti de la structure distribuée du système pour obtenir des techniques de vérification plus efficaces, ainsi que des formalismes de spécification mieux adaptés. L'objectif de cette partie sera de présenter ces techniques.

- Lorsque l'on parle de système séquentiel, le modèle naturel pour un comportement est un *mot*. La première question de cette partie à laquelle on s'intéresse est : comment représenter un comportement *concurrent* ? On utilise des graphes étiquetés ayant certaines propriétés. Deux exemples sont donnés ci-dessous.
  Le premier modèle étudié est celui des traces. Un exemple est donné en Fig. 1.2. Une trace est un ordre partiel dont chaque sommet est étiqueté par des noms d'actions (a,b,c sur la Fig. 1.2). On peut lire la Fig. 1.2 comme une exécution d'un système,

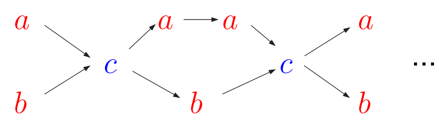


FIG. 1.2 – *Trace, dans laquelle les actions a et b sont concurrentes*

dans laquelle interviennent trois actions a, b et c. Les actions a et b sont concurrentes : on trouve en effet certains a en parallèle avec certains b. Par ailleurs, aucun a n'est directement relié à un b. On peut voir les actions a et b exécutées par deux processus différents p_1 et p_2 et ne nécessitant pas de partage de ressources. L'action c est directement reliée à des actions a et b. Ceci représente une dépendance entre c et a d'une part, et entre c et b d'autre part. On peut voir c exécutée par p_1 et p_2, en rendez-vous.
Un autre exemple est celui des MSC (Message Sequence Charts, ou diagrammes de séquence). Un MSC représente l'exécution d'un système composé de plusieurs processus, qui communiquent par des canaux fiables (FIFO ou non, suivant le cadre dans lequel on se place). Un exemple est donné en Fig. 1.3.

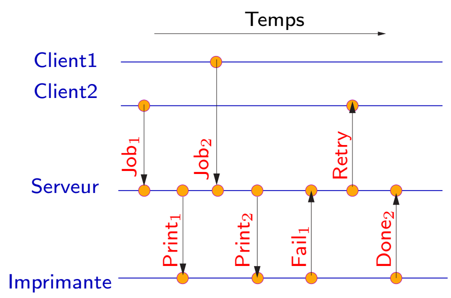


FIG. 1.3 – *MSC modélisant un scénario entre 4 processus*

- La difficulté suivante consiste à généraliser la modélisation d'un système séquentiel dans le cadre concurrent. Plusieurs généralisations existent : réseaux de Petri, automates distribués, High-level Message Sequence Charts (HMSC), automates communicants.
- Il faut ensuite envisager les formalismes (logiques, par exemple) permettant d'exprimer des propriétés de systèmes concurrents : logiques (temporelles) sur les traces, inclusion d'un langage de MSC représentant un système dans un autre représentant un ensemble de bons comportements,...
- Les principaux problèmes abordés seront ceux du model-checking et de la satisfaisabilité.

2. Vérification des systèmes séquentiels

2.1 Généralités et notations

On se fixe un ensemble fini de propriétés élémentaires $\mathscr{P}$ observables. En chaque état des systèmes que l'on étudie, on observe quelles propriétés de $\mathscr{P}$ sont vérifiées (les autres ne l'étant pas). Formellement, dans toute cette partie, un système $\mathscr{S}$ sera modélisé comme un système de transitions, avec propriétés observables : $\mathscr{S} = (S, S_0, \rightarrow, \mathscr{P}, \pi)$, où

- S est l'ensemble (fini) des états du système,
- S_0 est l'ensemble des états initiaux,
- $\rightarrow$ est la relation de transition : $\rightarrow \subseteq S \times S$,
- $\pi : S \rightarrow 2^{\mathscr{P}}$ étiquette chaque état $s \in S$ avec l'ensemble des propriétés $\pi(s)$ observables en s.

Un *run* est un chemin maximal $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ dans S ($s_0 \in S_0$). Un run représente une exécution du système (on peut ne s'intéresser qu'aux chemins finis, ou au contraire qu'aux chemins infinis). Il induit le mot infini $\pi(s_0)\pi(s_1)\pi(s_2)\cdots \in (2^{\mathscr{P}})^\omega$ sur lequel on vérifiera des propriétés. On appelle ce mot infini un *comportement* de $\mathscr{S}$.

Du point de vue de la vérification des systèmes séquentiels, on peut s'intéresser à deux types de propriétés. Celles dites du *temps linéaire (linear time)* permettent de spécifier des propriétés sur l'ensemble des comportements du système, qui sont des mots sur $2^{\mathscr{P}}$, sans s'intéresser aux choix que le système a pu effectuer, à la façon dont ils ont été produits. Celles dites du *temps arborescent (branching time)*, qui parlent de l'arbre de calcul du système.

Soient par exemple, les systèmes $\mathscr{S}_1$, $\mathscr{S}_2$ de la Fig. 2.1. On obtient comme unique

FIG. 2.1 – *Deux systèmes indifférenciés en temps linéaire, différenciés en temps arborescent*

exécution pour $\mathscr{S}_1$ comme pour $\mathscr{S}_2$ celle de la Fig 2.2. Les arbres d'exécutions représentés

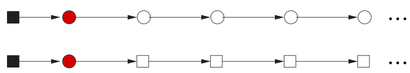


FIG. 2.2 – *Les exécutions de $\mathscr{S}_1$ et $\mathscr{S}_2$*

en Fig 2.3 sont par contre différentiables en logique temporelle arborescente CTL.

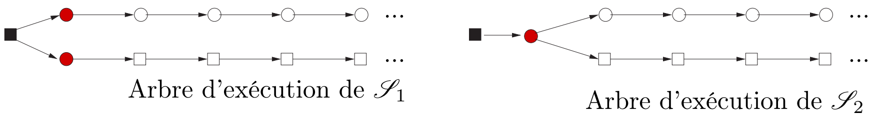


FIG. 2.3 – *Les arbres d'exécution de $\mathscr{S}_1$ et $\mathscr{S}_2$*

2.2 Logique temporelle linéaire LTL (Pnueli [25])

2.2.1 Syntaxe

Soit $\Sigma = 2^{\mathscr{P}}$. L'ensemble $\mathsf{LTL} = \mathsf{LTL}_\Sigma(\mathbf{X},\mathbf{U})$ est le plus petit ensemble de formules tel que

- $\mathscr{P} \subseteq \mathsf{LTL}_\Sigma$.
- $\alpha \in \mathsf{LTL}_\Sigma \Longrightarrow \neg\alpha \in \mathsf{LTL}_\Sigma$,
- $\alpha,\beta \in \mathsf{LTL}_\Sigma \Longrightarrow \alpha \vee \beta \in \mathsf{LTL}_\Sigma$,
- $\alpha \in \mathsf{LTL}_\Sigma \Longrightarrow \mathbf{X}\alpha \in \mathsf{LTL}_\Sigma$,
- $\alpha,\beta \in \mathsf{LTL}_\Sigma \Longrightarrow \alpha \,\mathbf{U}\, \beta \in \mathsf{LTL}_\Sigma$.

X se lit neXt et **U** se lit Until.

2.2.2 Sémantique

On va évaluer (*in fine*) les formules de $\mathsf{LTL}(\mathbf{X},\mathbf{U})$ sur des mots sur l'alphabet $\Sigma = 2^{\mathscr{P}}$:

- ou bien sur les mots finis : Σ^*, lorsqu'on s'intéresse aux programmes dont le comportement est fini.
- ou bien sur les mots infinis : Σ^ω, dans le cadre de la vérification des systèmes réactifs.

Soit $u = u_0u_1 \cdots \in \Sigma^*$ ou Σ^ω un mot (fini ou infini) et i une position dans le mot (les positions sont numérotées à partir de 0). Si u est fini, la dernière position de u est donc $|u| - 1$, où $|u|$ désigne la longueur de u. Soit α une formule de LTL. On définit la relation de satisfaction $u,i \models \alpha$, lue « u satisfait α à la position (ou au temps) i » (où $0 \leqslant i \leqslant |u| - 1$), de la façon suivante.

- ○ $u,i \models p$, pour $p \in \mathscr{P}$, si $p \in u_i$;
- ○ $u,i \models \alpha \vee \beta$ si $u,i \models \alpha$ ou $u,i \models \beta$;
- ○ $u,i \models \neg\alpha$ si l'on n'a pas $u,i \models \alpha$;
- ○ $u,i \models \mathbf{X}\alpha$ si $i + 1 \leqslant |u| - 1$ (pour les mots finis) et $u,i + 1 \models \alpha$;
- ○ $u,i \models \alpha \,\mathbf{U}\, \beta$ s'il existe un entier j qui satisfait les conditions suivantes :
  - $i \leqslant j \leqslant |u| - 1$,
  - $u,j \models \beta$,
  - pour tout k tel que $i \leqslant k \leqslant j - 1$, on a : $u,k \models \alpha$.

On dit qu'un mot u satisfait une formule α s'il la satisfait à l'instant 0, c'est-à-dire si $u,0 \models \alpha$. Soit α une formule de $\mathsf{LTL}(\mathbf{X},\mathbf{U})$. Les langages de mots finis ou infinis définis par α sont :

$$L_\Sigma(\alpha) = \{u \in \Sigma^* \mid u,0 \models \alpha\}$$
$$L^\omega_\Sigma(\alpha) = \{u \in \Sigma^\omega \mid u,0 \models \alpha\}$$

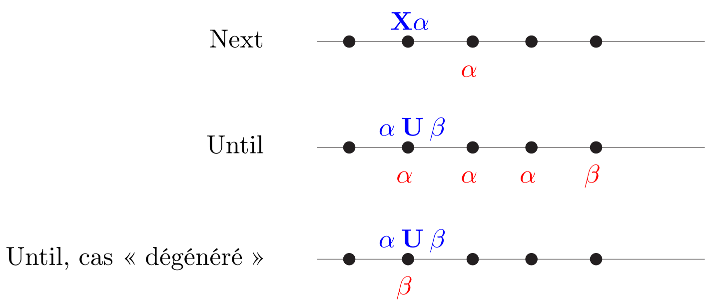


FIG. 2.4 – *Next et Until*

On dit qu'un langage L est *exprimable* (ou *définissable*) dans $\mathsf{LTL}(\mathbf{X},\mathbf{U})$ s'il existe une formule α de $\mathsf{LTL}(\mathbf{X},\mathbf{U})$ telle que $L = L_\Sigma(\alpha)$. On dit aussi que α *définit* L.

Remarques

- Toute propriété de LTL peut être décrite comme un automate de Büchi (cf. section 2.2.3).
- On utilise les abréviations usuelles tt pour $p \vee \neg p$ (vrai), ff pour $\neg\mathsf{tt}$ (faux), $\alpha \wedge \beta$ pour $\neg(\neg\alpha \vee \neg\beta)$, $\alpha \Rightarrow \beta$ pour $\neg\alpha \vee \beta$ et $\alpha \Leftrightarrow \beta$ pour $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
- Par définition, si β est vraie à un instant, $\alpha\,\mathbf{U}\,\beta$ l'est aussi pour tout α.
- Si $\alpha\,\mathbf{U}\,\beta$ est vraie un instant, β doit être vraie dans le futur (avoir toujours α ne suffit pas).
- Plus généralement, $\alpha\,\mathbf{U}\,\beta = \beta \vee (\alpha \wedge \mathbf{X}(\alpha\,\mathbf{U}\,\beta))$.
- La validité d'une formule de $\mathsf{LTL}(\mathbf{X},\mathbf{U})$ ne dépend que du futur : pour v fini et $\alpha \in \mathsf{LTL}(\mathbf{X},\mathbf{U})$, on a $vu,|v| \models \alpha$ ssi $u,0 \models \alpha$.

Exercice Exprimer les propriétés suivantes par des automates de Büchi et par des formules de LTL.

- La propriété p arrive un jour : $\mathsf{tt}\,\mathbf{U}\,p$. On utilise l'abréviation $\mathbf{F}\alpha \equiv \mathsf{tt}\,\mathbf{U}\,\alpha$.
- La propriété p est toujours vraie :$\neg\mathbf{F}\neg p$. On utilise l'abréviation $\mathbf{G}\alpha \equiv \neg\mathbf{F}\neg\alpha$.
- Le comportement est fini (si on évaluait les formules sur $\Sigma^* \cup \Sigma^\omega$).
- La propriété p est vraie à l'instant 1.
- Les propriétés p et q n'arrivent jamais en même temps : $\mathbf{G}(p \Rightarrow \neg q)$.
- Tout p est suivi immédiatement après par un p où un q.
- Vivacité : la propriété p est répétée infiniment souvent.
- Toute demande ressource est acquittée (dans le futur).
- Toute demande infiniment répétée finit par être acquittée.
- Équité forte : toute demande infiniment répétée est acquittée infiniment souvent.
- Équité faible : toute demande répétée de façon continue finit par être acquittée.
- La propriété p est stable (si elle arrive, elle demeure).
- Les langages a^ω, $(ab)^+$, $(ab)^\omega$ sont définissables
- Qu'en est-il pour $(aa)^+$, $(aa)^\omega$?

$\square$

Remarque Un résultat fondamental sur les langages de mots infinis est l'équivalence entre la définissabilité en LTL, logique du premier ordre, par expression rationnelle sans étoile, et reconnaissabilité par des monoïdes apériodiques.

Exercice – Montrer qu'on peut remplacer les deux modalités $\mathbf{X}$ et $\mathbf{U}$ par l'unique modalité $\mathbf{XU}$, définie par $\alpha\mathbf{XU}\beta \equiv \mathbf{X}(\alpha\,\mathbf{U}\,\beta)$, et que les deux modalités $\mathbf{X}$ et $\mathbf{U}$ se redéfinissent à partir de $\mathbf{XU}$.
- Vérifier que $\neg\mathbf{X}\alpha = \mathbf{X}\neg\alpha$, $\mathbf{X}(\alpha \vee \beta) = \mathbf{X}\alpha \vee \mathbf{X}\beta$.
- On peut définir le dual $\mathbf{R}$ (*release*) de la modalité $\mathbf{U}$, par $\alpha\,\mathbf{R}\,\beta = \neg(\neg\alpha\,\mathbf{U}\,\neg\beta)$. Vérifier que $\alpha\,\mathbf{R}\,\beta = \mathbf{G}\beta \vee (\beta\,\mathbf{U}\,(\alpha \wedge \beta))$.

□

2.2.3 Satisfaisabilité et model-checking : approche automates

Dans cette section, on se place dans le cadre de la vérification des comportements infinis. Pour une formule LTL α, on note $L(\alpha) = L^\omega(\alpha)$.

Le problème du model-checking est de vérifier si tous les comportements d'un système de transitions $\mathscr{S}$ dont les états vérifient des propriétés de $\mathscr{P}$ vérifient une formule $\alpha \in \mathsf{LTL}_{2^\mathscr{P}}$. L'idée est de transformer $\mathscr{S}$ en un automate $\mathscr{A}_\mathscr{S}$ qui accepte les observations de $\mathscr{S}$, puis de « compiler » la formule α en automate de mots infinis $\mathscr{A}_\alpha$, tel que $L(\alpha) = L(\mathscr{A}_\alpha)$. On peut de même construire $\mathscr{A}_{\neg\alpha}$. On construit par ailleurs facilement un automate (fini) qui accepte le comportement du système de transitions (fini) $\mathscr{S}$. Ces deux automates permettent de tester
- si $L(\mathscr{A}_\alpha) = \varnothing$, autrement dit, si la formule α est satisfaisable ;
- si $L(\mathscr{A}_\mathscr{S}) \cap L(\mathscr{A}_{\neg\alpha}) = \varnothing$, autrement dit, si tous les comportements du système de transitions $\mathscr{S}$ sont corrects vis-à-vis de α.

On note $\mathsf{alph}(\alpha)$ l'ensemble des propositions atomiques intervenant dans α. Comme α ne peut parler que des propositions de $\mathsf{alph}(\alpha)$, on vérifie facilement que α est satisfaisable s'il existe un mot de $2^{\mathsf{alph}(\alpha)}$ qui satisfait α (projeter chaque lettre d'un mot de $2^\mathscr{P}$ sur $2^{\mathsf{alph}(\alpha)}$, en effaçant les propositions de $\mathscr{P} \setminus \mathsf{alph}(\alpha)$, ne change pas la validité de α). On peut donc supposer que $\mathsf{alph}(\alpha) = \mathscr{P}$.

Automates de Büchi généralisés

Un automate de Büchi généralisé sur Σ est un automate $\mathscr{A} = (S, \rightarrow, S_0, F_1, \ldots, F_\ell)$. Par rapport à un automate sur les mots, la structure est la même, à ceci près qu'il y a ℓ sous-ensembles d'états finals $F_1, \ldots, F_l$. Lorsque $\ell = 1$, on dit que $\mathscr{A}$ est un automate de Büchi. Un *run* sur $u \in \Sigma^\omega$ est une suite infinie d'états $\sigma = s_0 s_1 \cdots \in S^\omega$ telle que $s_0 \in S_0$, et pour $i \geqslant 0$, $s_i \xrightarrow{u_i} s_{i+1}$, où $u = u_0 u_1 \cdots$, $u_i \in \Sigma$. L'ensemble des états infiniment répétés dans σ est $\inf(\sigma) = \Big\{ s \in S \mid \{i \geqslant 0 \mid s = s_i\} \text{ est infini} \Big\}$. Le run σ est acceptant si

$$\forall i \in \{1, \ldots, \ell\}, \quad \inf(\sigma) \cap F_i \neq \varnothing$$

Le langage $L(\mathscr{A})$ accepté par $\mathscr{A}$ est l'ensemble des étiquettes des runs acceptants.

On vérifie facilement qu'étant donnés deux automates de Büchi généralisés $\mathscr{A} = (S, \to ,S_0,F_1, \dots ,F_\ell)$ et $\mathscr{B} = (T, \to ,T_0,G_1, \dots ,G_k)$, on peut construire un automate reconnaissant leur intersection de taille $|\mathscr{A}| \cdot |\mathscr{B}|$: il suffit de faire le produit cartésien habituel de $\mathscr{A}$ et $\mathscr{B}$ avec comme famille d'ensemble d'états finals $(S \times G_1, \dots S \times G_\ell, F_1 \times T, \dots ,F_k \times T)$.

De même, tester si $L(\mathscr{A}) \neq \varnothing$ revient à tester si on a un chemin $s_0 \xrightarrow{+} t_1 \in F_1 \xrightarrow{+} t_2 \in F_2 \xrightarrow{+} \cdots \xrightarrow{+} t_\ell \in F_\ell \xrightarrow{+} t_1$. Il s'agit d'une propriété d'accessibilité, NLOGSPACE-complète. En utilisant l'algorithme de Tarjan, on obtient un algorithme déterministe linéaire.

Par contre, le complément et la déterminisation sont difficiles (construction de Safra, cf. [34, 27, 39, 23, 22]), et on ne peut pas exprimer tous les langages par des automates déterministes. Par exemple,

- On peut facilement construire un automate de Büchi reconnaissant les mots de $\{a,b\}^\omega$ ayant un nombre infini de a,
- On peut également construire un automate de Büchi reconnaissant les mots de $\{a,b\}^\omega$ ayant un nombre fini de a, mais on peut montrer qu'un tel automate n'est jamais déterministe. Par ailleurs, on trouve facilement pour ce langage deux automates non isomorphes avec un nombre d'états minimal (2, en l'occurrence) reconnaissant ce langage. Il n'y a donc pas la même notion d'automate minimal que dans les mots finis.

Calcul de l'automate $\mathscr{A}_\mathscr{S}$ acceptant les observations de $\mathscr{S}$

Si $\mathscr{S} = (S,S_0, \to ,\mathscr{P},\pi)$, il suffit de prendre $\mathscr{A}_\mathscr{S} = (S,S_0, \to ,\mathscr{P},S)$ avec $q \xrightarrow{a} p$ ssi on a $q \longrightarrow p$ dans $\mathscr{S}$, et $a = \pi(q)$. On vérifie que les mots acceptés par l'automate de Büchi $\mathscr{A}_\mathscr{S}$ sont bien les observations $\pi(k_0)\pi(k_1)\cdots$ des chemins dans $\mathscr{S}$.

Calcul de l'automate $\mathscr{A}_\alpha$ tel que $L(\mathscr{A}_\alpha) = L(\alpha)$

On peut pour une autre présentation, basée sur les automates alternants, commencer par le papier [40] de M. Vardi.

La clôture $\mathsf{cl}(\alpha)$ d'une formule α est, de façon informelle, l'ensemble des sous-formules de α et leur négation. Un état de l'automate $\mathscr{A}_\alpha$ sera un ensemble « consistant » de telles sous-formules. L'idée est que le langage des mots lus à partir d'un état $A \subseteq \mathsf{cl}(\alpha)$ est le langage des mots vérifiant exactement A parmi les formules de $\mathsf{cl}(\alpha)$.

- $\alpha \in \mathsf{cl}'(\alpha)$,
- $\neg\beta \in \mathsf{cl}'(\alpha) \Rightarrow \beta \in \mathsf{cl}'(\alpha)$
- $\beta \vee \gamma \in \mathsf{cl}'(\alpha) \Rightarrow \beta,\gamma \in \mathsf{cl}'(\alpha)$
- $\mathbf{X}\beta \in \mathsf{cl}'(\alpha) \Rightarrow \beta \in \mathsf{cl}'(\alpha)$
- $\beta \,\mathbf{U}\, \gamma \in \mathsf{cl}'(\alpha) \Rightarrow \mathbf{X}(\beta \,\mathbf{U}\, \gamma),\beta,\gamma \in \mathsf{cl}'(\alpha)$

Enfin, on clôt cl' par complément pour obtenir cl, en identifiant $\neg\neg\varphi$ et φ.

Remarque On a facilement $|\mathsf{cl}(\alpha)| = O(|\alpha|)$.

Un état q de $\mathscr{A}_\alpha$ est un sous ensemble de $\mathsf{cl}(\alpha)$ vérifiant les propriétés suivantes (consistance).

- $\forall\beta \in \mathsf{cl}(\alpha)$, $\beta \in q \Leftrightarrow \neg\beta \notin q$.

– $\forall \beta \vee \gamma \in \mathsf{cl}(\alpha)$, $\beta \vee \gamma \in q \Leftrightarrow \beta \in q$ ou $\gamma \in q$.
– $\forall \beta \,\mathbf{U}\, \gamma \in \mathsf{cl}(\alpha)$, $\beta \,\mathbf{U}\, \gamma \in q \Leftrightarrow \gamma \in q$ ou $\Big(\beta \in q$ et $\mathbf{X}(\beta \,\mathbf{U}\, \gamma) \in q\Big)$.

Les états initiaux sont ceux « vérifiant » α. Les transitions assurent que les transitions se font correctement vis-à-vis des propositions atomiques à vérifier ainsi que vis-à-vis de la modalité $\mathbf{X}$. Enfin, les conditions de Büchi généralisées assurent qu'un état qui promet $\varphi \,\mathbf{U}\, \psi$ finira par vérifier ψ.

– États initiaux $= \{q \mid \alpha \in q\}$.
– $q \xrightarrow{P} q'$ si $P = q \cap \mathsf{alph}(\alpha)$ et, pour tout $\mathbf{X}\beta \in \mathsf{cl}(\alpha)$, $\mathbf{X}\beta \in q \Leftrightarrow \beta \in q'$. Note : P peut être vide.
– Soient $\gamma_i \,\mathbf{U}\, \delta_i$ les formules Until apparaissant dans $\mathsf{cl}(\alpha)$. On ajoute une condition de Büchi généralisée pour chaque formule de ce type :

$$F_i = \{q \mid \neg(\gamma_i \,\mathbf{U}\, \delta_i) \in q \text{ ou } \delta_i \in q\}$$

Théorème 2.2.1 *Si $x \in 2^{\mathsf{alph}(\alpha)}$, $x \models \alpha \Longleftrightarrow x \in L(A_\alpha)$.*

Preuve. ($\Longrightarrow$) Soit $x = x_0 x_1 \cdots$ tel que $x \models \alpha$. On construit un run acceptant de $\mathscr{A}_\alpha$ sur x. On note $q_i = \{\beta \in \mathsf{cl}(\alpha) \mid x,i \models \beta\}$. On vérifie alors immédiatement que
– on a $\alpha \in q_0$;
– pour tout i, q_i est un état ;
– pour tout i, on a $q_i \xrightarrow{x_i} q_{i+1}$.

On en déduit que $\theta = q_0 q_1 \cdots$ est un run sur x. Il reste montrer qu'il est acceptant. Si ce n'était pas le cas, il existerait une formule $\beta_i \,\mathbf{U}\, \gamma_i$ de $\mathsf{cl}(\alpha)$ telle que $\inf(\theta) \cap F_i = \varnothing$. Donc, à partir d'un certain indice j_0, θ ne rencontre plus F_i. Ceci implique que $\neg(\beta_i \,\mathbf{U}\, \gamma_i) \notin q_n$ et $\gamma_i \notin q_n$ pour $n \geqslant j_0$, donc que $\beta_i \,\mathbf{U}\, \gamma_i \in q_n$ (par définition des états) et $\gamma_i \notin q_n$. Par définition de q_n, on a donc $x,j_0 \models \beta_i \,\mathbf{U}\, \gamma_i$ et $x,n \not\models \gamma_i$ pour $n \geqslant j_0$, contradiction.

($\Longleftarrow$) Soit $x \in L(\mathscr{A}_\alpha)$. Il existe un run acceptant de $\mathscr{A}_\alpha$ sur x, on le note $\theta = q_0 q_1 \cdots$. On montre par induction structurelle sur $\beta \in \mathsf{cl}(\alpha)$ que pour tout $i \geqslant 0$, on a

$$x,i \models \beta \text{ ssi } \beta \in q_i \tag{2.1}$$

La propriété (2.1) se vérifie facilement si β est une proposition atomique (par définition des transitions), si β est de la forme $\neg\gamma$ ou $\gamma \vee \delta$ (par induction et consistance des états). Si $\beta = \mathbf{X}\gamma$, alors, $x,i \models \beta$ ssi $x,i+1 \models \gamma$ ssi $\gamma \in q_{i+1}$ (par induction structurelle) ssi $\beta \in q_i$ (par définition des transitions).

Reste le cas $\beta = \gamma \,\mathbf{U}\, \delta$. Supposons que $x,i \models \beta$ et montrons que $\beta \in q_i$. Soit $k \geqslant i$ tel que $x,k \models \delta$ et pour $i \leqslant j < k$, $x,j \models \gamma$. Si $k = i$, le résultat s'obtient directement par induction structurelle, puisqu'alors $\delta \in q_i$, donc $\beta \in q_i$ par définition des états. Sinon, on obtient le résultat par induction sur $k - i$. Si $x,i \not\models \delta$, on a $x,i \models \gamma$ et $u,i \models \mathbf{X}\beta$, car β est équivalente à $\delta \vee \gamma \wedge \mathbf{X}\beta$. Donc $x,i \models \gamma$ et $u,i+1 \models \beta$. Par induction structurelle, $\gamma \in q_i$. Par induction sur $k - i$, $\beta \in q_{i+1}$. Par définition des transitions, $\mathbf{X}\beta \in q_i$. Par définition des états, q_i, qui contient γ et $\mathbf{X}\beta = \mathbf{X}(\gamma \,\mathbf{U}\, \delta)$, contient aussi β.

Réciproquement, supposons que $\beta \in q_i$ et montrons que $x,i \models \beta$. Comme β est une formule Until, $\beta = \gamma_j \,\mathbf{U}\, \delta_j$ pour un certain j et il lui correspond la condition de Büchi

$F_j = \{q \mid \neg\beta \in q \text{ ou } \delta \in q\}$. Comme le run sur x est acceptant, il passe infiniment souvent par F_j. Soit k le plus petit indice après i dans le run tel que $q_k \in F_j$. Si $k = i$, $q_i \in F_j$, et puisque $\beta \in q_i$, $\neg\beta \notin q_i$, donc $\delta \in q_i$: on peut conclure par induction structurelle que $x,i \models \delta$, donc $x,i \models \beta$. Si $k > i$, on raisonne par induction sur $k - i$. On a $\beta \in q_i$, et $\delta \notin q_i$ (sinon, on serait dans un état de F_j). Par construction des états, on a dans q_i les propriétés $\mathbf{X}\beta$ et γ. Par induction structurelle, $x,i \models \gamma$. Par définition des transitions, $\beta \in q_{i+1}$. Par induction sur $k-i$, $x,i+1 \models \beta$, donc $x,i \models \mathbf{X}\beta$. Finalement, $x,i \models \gamma \wedge \mathbf{X}\beta$, donc $x,i \models \beta$.

En utilisant (2.1) et la définition des états initiaux, on obtient $x \models \alpha$. □

En utilisant le calcul de l'intersection d'automates de Büchi généralisés, le test du vide (linéaire en déterministe et NLOGSPACE), et le fait que l'automate $\mathscr{A}_\alpha$ est de taille $2^{O(|\alpha|)}$, on obtient :

Théorème 2.2.2 *La satisfaisabilité d'une formule α de* LTL *peut se tester en temps $2^{O(|\alpha|)}$.*

Le model-checking de $\mathscr{S}$ vis-à-vis de $\alpha \in$ LTL se teste en $|\mathscr{S}|2^{O(|\alpha|)}$.

Ces deux problèmes sont dans PSPACE.

2.2.4 Satisfaisabilité et model-checking : bornes inférieures

Pour les résultats de cette partie, voir [36] et [11]. Le but de cette partie est de montrer le théorème suivant.

Théorème 2.2.3 *La satisfaisabilité d'une formule de* LTL($\mathbf{U}$,$\mathbf{X}$), *SAT*($\mathbf{U}$,$\mathbf{X}$) *et le model-checking MC*($\mathbf{X}$,$\mathbf{U}$) *sont* PSPACE-*difficiles (donc* PSPACE-*complets).*

On commence par la réduction suivante (de façon analogue, le problème d'évaluer un circuit Booléen pour une valuation donnée est plus facile que celui de déterminer l'existence d'une valuation satisfaisante).

Théorème 2.2.4 *MC*($\mathbf{U}$,$\mathbf{X}$) $\leqslant_P$ *SAT*($\mathbf{U}$,$\mathbf{X}$)

Preuve. Soit $\mathscr{S} = (S, s_0, \to, \pi)$ un système de transitions et α une formule. On se place dans le cas du *model-checking* positif : on se demande s'il existe un chemin dans $\mathscr{S}$ qui satisfait α (ce qui revient au même vis-à-vis de la complexité, il suffit de changer α et $\neg\alpha$). On veut construire une formule ψ de taille polynomiale, telle que ψ est satisfaisable si et seulement s'il existe une exécution dans $\mathscr{S}$ qui satisfait α.

On se place sur un alphabet de propositions plus grand. On considère pour chaque $s \in S$ une nouvelle proposition atomique q_s et on définit les formules

$$\vartheta_s = \bigwedge_{p \in \pi(s)} p \wedge \bigwedge_{p \notin \pi(s)} \neg p \wedge \mathbf{X} \bigvee_{t \mid s \to t} q_t$$

et

$$\eta = \mathbf{G}\left(\bigvee_{s \in S} \vartheta_s \wedge \bigwedge_{s \neq t} (q_s \Rightarrow \neg q_t) \wedge \bigvee_{s \in S} q_s\right)$$

Finalement, soit $\psi = \alpha \wedge \eta \wedge q_{s_0}$. On vérifie alors facilement qu'il existe une exécution de $\mathscr{S}$ qui satisfait α si et seulement si ψ est satisfaisable. La réduction est clairement polynomiale. □

Il reste à montrer que le model-checking est PSPACE-difficile. On réduit **QBF** à MC.
Problème **QBF**:

Donnée Une formule $\gamma = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \gamma_0$, où $\gamma_0 = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{r_i} a_{i,j}$, où chaque Q_i est un quantificateur et où $a_{i,j}$ est une variable ou sa négation.

Question γ est-elle valide ?

Théorème 2.2.5 ***QBF*** *est PSPACE-complet.*

Proposition 2.2.6 *γ est valide s'il existe un ensemble $\mathscr{V}$ de valuations de variables, tel que*

- *$\mathscr{V}$ est valide : $\forall x \in \mathscr{V}$, $x \models \gamma_0$.*
- *$\mathscr{V}$ est clos : $\forall x \in \mathscr{V}$, si $Q_i = \forall$, $\exists y \in \mathscr{V}$, $y_{<i} = x_{<i}$ et $y_i = \neg x_i$.*

Théorème 2.2.7 ***QBF*** $\leqslant_P$ *MC(U).*

Preuve. Voir [11]. À γ, on associe un système de transitions $\mathscr{S}_\gamma$ de taille polynomiale. Pour chaque variable x_i, on associe un sous-système g_i: $s_i \rightarrow x_i^t, x_i^f \rightarrow e_i$. On ajoute un sommet e_0. On relie e_0 à s_1, e_1 à s_2, etc. Arrivé e_n, on continue en enchaînant m sous-systèmes, un pour chaque disjonction de γ_0. Le premier état du premier est $e_n = f_1$, le $i^{\text{ème}}$ est de la forme $f_i \rightarrow a_{i,1}, \ldots, a_{i,k_i} \rightarrow f_{i+1}$. De l'état f_{m+1}, on peut retourner dans s_i.

On a une notion de valuation courante lorsqu'on est dans la $2^{\text{ème}}$ partie de $\mathscr{S}_\gamma$. Un chemin (infini) ξ définit donc un ensemble de valuations $\mathscr{V}(\xi)$.

Une condition suffisante pour qu'un chemin définisse un ensemble de valuations clos, est que, pour un x_j quantifié universellement, si le chemin visite e_{j-1}, alors il visite à la fois x_j^tet x_j^f avant une prochaine (éventuelle) visite à e_{j-1}. Ceci peut s'écrire

$$\psi_j = \mathbf{G}(e_{j-1} \Rightarrow ((\neg s_{j-1}\, \mathbf{U}\, x_j^t) \wedge (\neg s_{j-1}\, \mathbf{U}\, x_j^f))) \text{ et } \psi_{\text{clo}} = \bigwedge_{Q_j = \forall} \psi_j$$

Les exécutions ξ satisfaisant ψ_{clos} sont telles que $\mathscr{V}(\xi)$ est clos.

On peut aussi imposer que lorsque le chemin ξ rencontre un état $a_{i,j}$, la valuation courante satisfait $a_{i,j}$ tel que défini dans la formule (x_k ou $\neg x_k$).

$$\psi_{i,j} = \mathbf{G}\Big(x_k^f \Rightarrow \mathbf{G}\neg a_{i,j} \vee \neg a_{i,j}\, \mathbf{U}\, s_k\Big) \qquad \text{si } a_{i,j} = x_k$$
$$\psi_{i,j} = \mathbf{G}\Big(x_k^t \Rightarrow \mathbf{G}\neg a_{i,j} \vee \neg a_{i,j}\, \mathbf{U}\, s_k\Big) \qquad \text{si } a_{i,j} = \neg x_k$$

On définit $\psi_{\text{corr}} = \bigwedge_{i,j} \psi_{i,j}$ et $\alpha_\gamma = \psi_{\text{clo}} \wedge \psi_{\text{corr}}$. On vérifie alors que $\mathscr{S}$ vérifie α_γ si et seulement si γ est valide, et que $\mathscr{S}_\gamma$ et α_γ sont de tailles polynomiales. □

Corollaire 2.2.8 *MC(U) et SAT(U) sont* ***PSPACE****-difficiles.* □

2.3 Logiques arborescentes

Les logiques arborescentes permettent de spécifier des propriétés de plusieurs exécutions d'un système [33, 9].

2.3.1 Syntaxe de CTL^* et CTL

Les formules de CTL^* sont définies
- $\mathscr{P} \subseteq \mathsf{CTL}^*_\Sigma$.
- $\alpha \in \mathsf{CTL}^*_\Sigma \Longrightarrow \neg\alpha \in \mathsf{CTL}^*_\Sigma$,
- $\alpha,\beta \in \mathsf{CTL}^*_\Sigma \Longrightarrow \alpha \vee \beta \in \mathsf{CTL}^*_\Sigma$,
- $\alpha \in \mathsf{CTL}^*_\Sigma \Longrightarrow \mathbf{X}\alpha \in \mathsf{CTL}^*_\Sigma$,
- $\alpha,\beta \in \mathsf{CTL}^*_\Sigma \Longrightarrow \alpha \,\mathbf{U}\, \beta \in \mathsf{CTL}^*_\Sigma$.
- $\alpha \in \mathsf{CTL}^*_\Sigma \Longrightarrow \mathbf{A}\alpha \in \mathsf{CTL}^*_\Sigma$.
- $\alpha \in \mathsf{CTL}^*_\Sigma \Longrightarrow \mathbf{E}\alpha \in \mathsf{CTL}^*_\Sigma$.

2.3.2 Sémantique de CTL^*

Les modèles des formules de CTL^* sont les systèmes de transitions. Étant donné un système $\mathscr{S} = (S,S_0, \rightarrow, \mathscr{P},\pi)$ et un run infini $\sigma = s_0s_1s_2\cdots$ de ce système, on définit la relation de satisfaction $\models$ par
- ○ $\sigma,i \models p$, pour $p \in \mathscr{P}$, si $p \in \pi(s_i)$;
- ○ $\sigma,i \models \alpha \vee \beta$ si $\sigma,i \models \alpha$ ou $\sigma,i \models \beta$;
- ○ $\sigma,i \models \neg\alpha$ si l'on n'a pas $\sigma,i \models \alpha$;
- ○ $\sigma,i \models \mathbf{X}\alpha$ si $\sigma,i+1 \models \alpha$;
- ○ $\sigma,i \models \alpha \,\mathbf{U}\, \beta$ s'il existe un entier j qui satisfait les conditions suivantes :
  - $i \leqslant j$,
  - $\sigma,j \models \beta$,
  - pour tout k tel que $i \leqslant k \leqslant j-1$, on a : $\sigma,k \models \alpha$.
- ○ $\sigma,i \models \mathbf{A}\alpha$ si pour tout run $\tau = t_0t_1\cdots$ tel que $s_0\cdots s_i = t_0\cdots t_i$, on a $\tau,i \models \alpha$.
- ○ $\sigma,i \models \mathbf{E}\alpha$ s'il existe un run $\tau = t_0t_1\cdots$ tel que $s_0\cdots s_i = t_0\cdots t_i$, et $\tau,i \models \alpha$.

$\mathscr{S} = (S,S_0, \rightarrow, \mathscr{P},\pi)$ satisfait α si $\sigma,0 \models \alpha$ pour tout run σ issu de S_0 (qui est en général un singleton).

Les opérateurs $\mathbf{A}$ et $\mathbf{E}$ quantifient sur les runs. Un run étant fixé, les opérateurs $\mathbf{X}$ et $\mathbf{U}$ quantifient sur les positions de ce run (comme en LTL). On utilise les abréviations usuelles $\mathbf{F}$,$\mathbf{G}$,$\mathbf{R}$...

On peut exprimer des propriétés naturelles en CTL^* :
- la propriété p est constamment vraie : $\mathbf{AG}p$.
- la propriété p est atteignable : $\mathbf{EF}p$.
- la propriété p est inévitable : $\mathbf{AF}p$.
- la propriété p est constamment atteignable : $\mathbf{AG\,E\,F}p$ (mais p n'est pas nécessairement inévitable).
- sur tout run, la propriété p arrive infiniment souvent : $\mathbf{AGF}p$.

2.3.3 Syntaxe de CTL

CTL est le fragment de CTL^* qui impose à tout opérateur temporel d'être sous la portée d'un quantificateur de runs $\mathbf{A}$ ou $\mathbf{E}$. Pour cette raison, on utilise les opérateurs

PQ, où **P** = **A** ou **E** et où **Q** est un opérateur de LTL. Par exemple $\alpha\,\mathbf{AU}\,\beta = \mathbf{A}(\alpha\,\mathbf{U}\,\beta)$. Ainsi, toute formule s'exprime en utilisant les opérateurs **AX**, **EX**, **AU** et **EU**.

La formule **AGF**p n'est pas, par exemple une formule de CTL (et on peut montrer qu'elle n'a pas de formule équivalente en CTL). Il y a par contre des formules de CTL$^*\setminus$CTL qui peuvent quand même s'exprimer en CTL. La validité d'une formule de CTL ne peut dépendre que du premier état du run sur lequel elle s'évalue.

On vérifie que $\neg\,\mathbf{AX}\,\alpha = \mathbf{EX}\neg\alpha$, que $\neg(\alpha\,\mathbf{AU}\,\beta) = (\neg\alpha)\,\mathbf{ER}\,(\neg\beta)$ (on a aussi d'ailleurs $\neg(\alpha\,\mathbf{EU}\,\beta) = (\neg\alpha)\,\mathbf{AR}\,(\neg\beta)$). Comme l'opérateur **R** s'exprime en fonction de **G** et **U**, on peut exprimer toute formule en utilisant les opérateurs **EX**, **EG** et **EU**. Cette traduction est exponentielle en la taille de la formule car l'expression $\alpha\,\mathbf{R}\,\beta = \mathbf{G}\beta \vee (\beta\,\mathbf{U}\,(\alpha \wedge \beta))$ fait apparaître 3 occurrences de β.

2.3.4 Model-checking de CTL

On se donne un système de transitions $\mathscr{S} = (S,\{s_0\}, \rightarrow, \mathscr{P},\pi)$ et une formule α de CTL, écrite avec les opérateurs **EX**, **EG** et **EU**, et on veut vérifier si $\mathscr{S}$ satisfait α.

On utilise le fait que la validité d'une formule de CTL ne dépend que du premier état du run sur lequel elle s'évalue : on marque chaque état s de $\mathscr{S}$ avec chaque sous-formule β de α ou sa négation, suivant que s satisfait β ou non. Le marquage se fait inductivement, en partant de celui donné par π pour les propositions atomiques. Étant donnés les marquages de β et γ, on doit donc calculer

- le marquage de $\neg\beta$ et $\beta \vee \gamma$: c'est trivial et linéaire en $|\mathscr{S}|$.
- le marquage de $\mathbf{EX}\beta$. Par définition, un état est marqué par $\mathbf{EX}\beta$ s'il a un successeur qui satisfait β. Le calcul de $\mathbf{EX}\beta$ se fait donc simplement par parcours de $\mathscr{S}$ en temps linéaire en $|\mathscr{S}|$.
- le marquage de $\beta\,\mathbf{EU}\,\gamma$. Un état satisfait cette formule s'il existe un chemin dont les états satisfont β amenant à γ. On peut se ramener facilement à un problème d'accessibilité sur un sous graphe de celui de $\mathscr{S}$, et le calcul du marquage de $\beta\,\mathbf{EU}\,\gamma$ est à nouveau linéaire en $|\mathscr{S}|$.
- le marquage de $\mathbf{EG}\beta$. Un état s satisfait cette formule si l'on peut atteindre de s une composante fortement connexe maximale dont tous les états satisfont β. On utilise ici encore l'algorithme de Tarjan, qui donne une réponse linéaire à ce problème.

Si l'on était parti d'une formule contenant l'opérateur **AU**, l'algorithme reste linéaire malgré l'utilisation de $\alpha\,\mathbf{R}\,\beta = \mathbf{G}\beta \vee (\beta\,\mathbf{U}\,(\alpha \wedge \beta))$ qui fait apparaître plusieurs occurrences de β (car on ne calcule le marquage de β qu'une seule fois). Un algorithme direct est donné dans [7].

Comme il y a un nombre linéaire de sous-formules de α (et négations de sous-formules), on en déduit:

Théorème 2.3.1 *Le model-checking de $\mathscr{S}$ vis-à-vis de α se teste en $O(|\mathscr{S}||\alpha|)$.*

3. Vérification des systèmes concurrents

Dans cette section, on présente les traces de Mazurkiewicz [26, 14, 20], un modèle dans lequel la concurrence est explicitement présente.

3.1 Les traces de Mazurkiewicz

Pour des détails sur cette section, voir [13, 14]. On fixe dans cette section un alphabet Σ. Une relation $\mathrm{D} \subseteq \Sigma \times \Sigma$ est une *relation de dépendance* si elle est réflexive et symétrique. Une *relation d'indépendance* est le complémentaire d'une relation de dépendance, *ie* une relation irréflexive et symétrique. Un couple (Σ,D), où D est une relation de dépendance, est appelé *alphabet de dépendance*. On notera I la relation d'indépendance $\Sigma^2 \setminus \mathrm{D}$. On dira que deux langages $L_1,L_2 \subseteq \Sigma^*$ sont indépendants (noté $L_1 \mathrm{I} L_2$) lorsque toute lettre d'un mot de L_1 est indépendante de toute lettre d'un mot de L_2.

Exemple 3.1.1 (Algorithme de Peterson) *Le code ci-dessous montre un algorithme d'exclusion mutuelle entre deux processus 0 et 1 communiquant par variables partagées (`turn`, `req`$_0$, `req`$_1$) et la relation de dépendance entre les actions (sur la figure, les dépendances entre actions d'un même processus ne sont pas toutes représentées).*

```
        int req0=0, req1=0;
        int turn;

        while(1)
        {
nci        <section non critique >
ri         reqi = true;
ti         turn = 1 - i;
           while (req1-i && (turn ≠ i))
                      tri          tti
               continue ;
eni        <section critique >
exi        reqi = 0;
        }
```

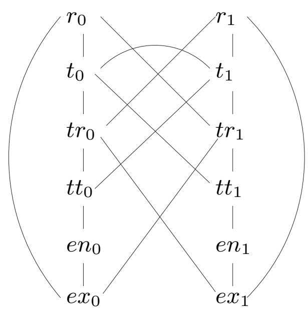


FIG. 3.1 – *Algorithme de Peterson*

Exemple 3.1.2 (cographe) *La famille des* cographes *est la plus petite famille d'alphabets de dépendance contenant les singletons, et stables par*

- *union disjointe : si* (Σ_1, D_1) *et* (Σ_2, D_2) *sont des cographes tels que* $\Sigma_1 \cap \Sigma_2 = \varnothing$, $(\Sigma_1 \cup \Sigma_2, \mathrm{D}_1 \cup \mathrm{D}_2)$ *est un cographe ;*
- *produit direct : si* (Σ_1, D_1) *et* (Σ_2, D_2) *sont des cographes tels que* $\Sigma_1 \cap \Sigma_2 = \varnothing$, $(\Sigma_1 \cup \Sigma_2, \mathrm{D}_1 \cup \mathrm{D}_2 \cup (\Sigma_1 \times \Sigma_2) \cup (\Sigma_2 \times \Sigma_1))$ *est un cographe.*

On peut montrer que les cographes sont les alphabets dont la relation de dépendance ne contient pas un sous graphe induit de la forme a—b—c—d (en omettant la réflexivité). Si $\leqslant$ est une relation d'ordre (partiel), on note $\lessdot = < \setminus <^2$ la relation successeur, ie, $x \lessdot y$ ssi $x < y$ et $\forall z, \quad x \leqslant z \leqslant y \Rightarrow z = x$ ou $z = y$.

Définition 3.1.1 (trace) *Soit* Σ, D *un alphabet de dépendance* *fixé*. *Une* trace *finie* $t = (v, \leqslant, \lambda)$ *sur* (Σ, D) *est donnée par un ensemble fini de sommets* V, *un ordre partiel* $\leqslant$ *sur* V *et un étiquetage* $\lambda : V \to \Sigma$ *des sommets de* t, *tels que*

– *si* $x \lessdot y$, *alors* $\lambda(x)\ \mathrm{D}\ \lambda(y)$.
– *si* $\lambda(x)\ \mathrm{D}\ \lambda(y)$, *alors* $x \leqslant y$ *ou* $y \leqslant x$.

On note $\mathbb{M}(\Sigma, \mathrm{D})$ *l'ensemble des traces sur l'alphabet de dépendance* (Σ,D).

Le fait que D soit réflexive implique que deux sommets étiquetés a ne peuvent pas être concurrents (ils doivent être ordonnés).

Exemple 3.1.3 *Le monoïde libre* Σ^* *est un monoïde de traces pour la relation de dépendance* $\mathrm{D} = \Sigma \times \Sigma$. *À l'inverse, le monoïde commutatif libre* $\mathbb{N}^k$ *est isomorphe à* $\mathbb{M}(\Sigma, \mathrm{D})$, *où* $|\Sigma| = k$ *et où* D *est l'égalité. On verra de façon générale que* $\mathbb{M}(\Sigma, \mathrm{D})$ *est naturellement muni d'une structure de monoïde, l'élément neutre étant la trace vide.*

Exemple 3.1.4 *Soit* Σ *un alphabet et* $\bar{\Sigma}$ *en bijection avec* Σ. *Pour chaque lettre* $a \in \Sigma$, *on a une copie* $\bar{a}$ *de* a *dans* $\bar{\Sigma}$. *La relation* $\mathrm{D} = (\Sigma \times \Sigma) \cup (\bar{\Sigma} \times \bar{\Sigma})$ *est une relation de dépendance sur* $A = (\Sigma \cup \bar{\Sigma})$.

Un sommet de t *étiqueté par une lettre de* Σ *n'est pas comparable avec un sommet de* t *étiqueté par une lettre de* $\bar{\Sigma}$ *: si c'était le cas, il existerait deux sommets consécutifs indépendants. Il en résulte qu'une trace* t *sur* A *est en bijection avec un couple de mots* $(u,v) \in \Sigma^* \times \bar{\Sigma}^*$.

On représente en général les traces par leur diagramme de Hasse, i.e. par le graphe de la relation $\lessdot$. Les notions de concaténation ainsi que celle de préfixe d'un mot, utilisée pour la sémantique de LTL pour représenter le temps discret, se généralisent facilement aux traces.

Exemple 3.1.5 (Algorithme de Peterson, suite) *Les traces suivantes représentent le début d'exécution de l'algorithme de Peterson.*

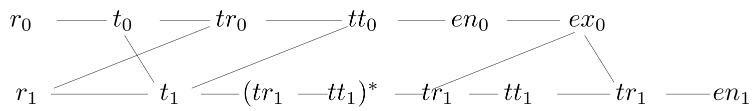


FIG. 3.2 – *Une exécution de l'algorithme de Peterson*

On voit que le deuxième processus est retardé jusqu'à ce que le premier sorte de sa section critique. On peut remarquer qu'un processus ne peut pas entrer dans sa section critique deux fois de suite si l'autre processus a également demandé d'y entrer (l'algorithme est équitable).

Définition 3.1.2 (concaténation, préfixe) *La* concaténation *de deux traces* $t_i = (V_i,\leqslant_i,\lambda_i)$, $i = 1,2$, *est* $t = t_1 t_2 = (V,\leqslant,\lambda)$ *avec* $V = V_1 \uplus V_2$, $\lambda_{|V_i} = \lambda_i$, *et* $\leqslant$ *est la clôture transitive de*

$$\leqslant_1 \cup \leqslant_2 \cup \{(x,y) \mid x \in V_1,\, y \in V_2,\ \text{et } \lambda(x)\ \mathrm{D}\ \lambda(y)\}$$

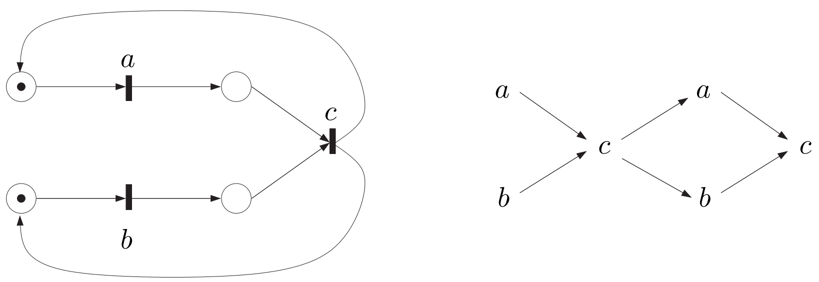
On vérifie que muni de la concaténation, $\mathbb{M}(\Sigma, \mathrm{D})$ *est un monoïde.*

Un préfixe *(ou* configuration*) u d'une trace t est donné par un sous-ensemble W de sommets de V fermé par le passé, i.e.,* $\forall x \in W$, $\forall y \in V$, $(y \leqslant x \implies y \in W)$. *Muni de l'ordre et de l'étiquetage induits, u est une trace. On vérifie que les préfixes de t sont exactement les traces u pour lesquelles il existe v,* $uv = t$.

Deux traces $x,y \in \mathbb{M}(\Sigma,\mathrm{D})$ admettent un *sup*, noté $x \vee y$ si elles sont préfixes d'une même trace. Leur sup est alors la plus petite trace dont elles sont toutes deux préfixes. Un langage de $\mathbb{M}(\Sigma,\mathrm{D})$ est *complet* s'il contient le sup (s'il existe) de deux quelconques de ses éléments.

Exemple 3.1.6 *Les traces peuvent servir à représenter le comportement d'un réseau de Petri 1-sûr, comme dans la Fig. 3.3 (a). Pour une transition t, on note* $^\bullet t$ *(resp.* $t^\bullet$*) l'ensemble des places lues par t (resp. sur lesquelles t écrit). Les trois transitions de ce réseau sont a, b, c. Soit* $\Sigma = \{a,b,c\}$ *et* I$= \{(a,c),(c,a)\}$, ie, D$= a$—b—c.

Un comportement de ce réseau est représenté sous forme de trace Fig 3.3 (b). Lorsqu'on observe le réseau, on peut voir les comportements abcabc et bacbac à partir du marquage initial de la Fig 3.3 (a). Mais ces deux comportements devraient être considérées comme équivalents du point de vue de la sémantique du réseau, car les transitions a et b sont indépendantes : $(^\bullet a \cup a^\bullet) \cap (^\bullet b \cup b^\bullet) = \varnothing$. *Ils ne se distinguent que par le point de vue de l'observateur. Une trace distingue l'indépendance entre actions contrairement à un mot, qui ne permet pas de savoir si deux actions consécutives dans le mot* doivent *être ordonnées causalement, ou apparaissent l'une après l'autre juste du fait d'une observation du système.*



(a) *(b)*

FIG. 3.3 – *(a) Un réseau de Petri (b) Une exécution de ce réseau*

De façon plus générale, les comportements d'un réseau élémentaire N (dont la sémantique impose aux places de contenir au plus un jeton) peuvent être représentés naturellement comme un langage de traces complet L_N. *De plus, deux linéarisations d'une même trace de* L_N *conduisent au même marquage dans le réseau, ce qui justifie la sémantique de ces réseaux par langages de traces.*

Définition 3.1.3 (Automates asynchrones)
Un automate asynchrone à k composants $\mathcal{A} = \langle (S_i)_{1\leqslant i\leqslant k},\Sigma,R,W,(\xrightarrow{a})_{a\in\Sigma},S^0,F\rangle$ *est constitué*
- *d'un ensemble d'états locaux* S_i *pour chaque composant* i,
- *d'un ensemble d'actions* Σ. *Chaque action* a *de* Σ *a un domaine de lecture* $R(a) \subseteq \{1,\dots,k\}$ *et un domaine d'écriture* $W(a) \subseteq \{1,\dots,k\}$, *tel que* $W(a) \subseteq R(a)$,
- *d'un ensemble de transitions* $\xrightarrow{a} \subseteq \prod_{i\in R(a)} S_i \times \prod_{i\in W(a)} S_i$,
- *d'un ensemble d'états initiaux* $S^0 \subseteq \prod_i S_i$,
- *d'un ensemble d'états finals* $F \subseteq \prod_i S_i$.

Exemple 3.1.7 (Algorithme de Peterson (suite)) *On peut décrire l'algorithme de Peterson sous forme d'automate asynchrone déterministe. Les composants sont les suivants :* S_i $(i = 0,1)$ *représentant les états de contrôle du processus* i, S_t *représentant les valeurs de la variable* `turn`, *et* S_i^r *représentant les valeurs des variables* $\mathtt{req}_i$. *Les actions sont celles données dans la Fig. 3.1. Par exemple, l'action* $tt_1 : S_1 \times S_t \to S_1$ *modifie juste le point de contrôle du processus 1. L'action* $ex_1 : S_1 \times S_1^r \to S_1 \times S_1^r$ *modifie le point de contrôle (retour au début de la boucle principale) ainsi que la valeur de* $\mathtt{req}_1$.

Remarque Deux actions $a,b \in \Sigma$ sont indépendantes si $R(a) \cap W(b) = \varnothing$ et $R(b) \cap W(a) = \varnothing$.

3.1.1 Traces et mots

L'ensemble des linéarisations de t est $\mathrm{Lin}(t) = \{\lambda(x_1)\cdots\lambda(x_n) \mid x_i \in V \text{ et } x_i < x_j \Rightarrow i < j\}$. On considère la congruence engendrée par $ab \sim ba$, $a \mathrel{\mathrm{I}} b$. On vérifie facilement que $\mathrm{Lin}(t)$ est une classe d'équivalence de $\sim$. On pourra représenter t par un de ses représentants x ou par cette classe (notée $\bar{x}$). Autrement dit, parce que l'ordre est contraint par l'étiquetage, on peut représenter une trace par un ensemble de mots. On vérifie que u est un préfixe de t ssi $\mathrm{Lin}(u)$ est l'ensemble des préfixes de $\mathrm{Lin}(t)$.

La projection canonique $\eta : \Sigma^* \to \mathbb{M}(\Sigma,\mathrm{D})$ est le morphisme de monoïdes qui envoie $a \in \Sigma$ sur la trace à un sommet étiquetée par a. On note $[x] = \eta(x)$, et on a $\bar{x} = \eta^{-1}(\eta(x))$. La trace $[x]$ est la seule trace ayant x comme linéarisation. Pour $L \subseteq \Sigma^*$, on note $[L] = \bigcup_{x\in L}[x]$ et $\bar{L} = \bigcup_{x\in L}\bar{x}$ est la *clôture par commutation de* L. On voit immédiatement que $\bar{L}$ est le plus petit langage contenant L et saturé par $\sim$.

3.1.2 Problèmes de décision

Dans le cadre de la vérification, il est important de savoir décider les problèmes d'intersection et d'inclusion.

Définition 3.1.4 *Soit* $L \subseteq \mathbb{M}(\Sigma,\mathrm{D})$. *On rappelle que*
- L *est* rationnel *si* $L = [K]$, *où* $K \subseteq \Sigma^*$ *est rationnel. Autrement dit,* L *est donné par une expression rationnelle sur* $\mathbb{M}(\Sigma,\mathrm{D})$.
- L *est* reconnaissable *si* $L = [K]$, *où* $K \subseteq \Sigma^*$ *est rationnel reconnu par un automate* I-*diamant (un automate est* I-*diamant si, pour tous* $a \mathrel{\mathrm{I}} b$, *si* $q \xrightarrow{ab} q'$ *alors* $q \xrightarrow{ba} q'$*).*

On note $\mathsf{Rec}(\mathbb{M}(\Sigma,\mathrm{D}))$ *et* $\mathsf{Rat}(\mathbb{M}(\Sigma,\mathrm{D}))$ *l'ensemble des langages reconnaissables (resp. rationnels) sur* $\mathbb{M}(\Sigma,\mathrm{D})$.

Comme $\mathbb{M}(\Sigma,\mathrm{D})$ est engendré par un nombre fini d'éléments (par $\eta(\Sigma)$), on a $\mathsf{Rec}(\mathbb{M}(\Sigma,\mathrm{D})) \subseteq \mathsf{Rat}(\mathbb{M}(\Sigma,\mathrm{D}))$. Par ailleurs, on peut vérifier que $L \subseteq \mathbb{M}(\Sigma,\mathrm{D})$ est reconnaissable ssi $\eta^{-1}(L) \subseteq \Sigma^*$ l'est. Pour $L \in \mathsf{Rat}(\Sigma^*)$, $\bar{L}$ n'est pas nécessairement rationnel sur Σ^* : soit $\Sigma = \{a,b\}$ avec $a \mathrel{\mathrm{I}} b$ et $L = (ab)^*$, on a alors $\bar{L} = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$.

Les problèmes de décision habituels sont décidables sur les reconnaissables mais pas en général sur les rationnels.

Théorème 3.1.1 *Soient* Σ_1,Σ_2 *des alphabets à au moins deux lettres disjoints,* $\Sigma = \Sigma_1 \cup \Sigma_2$ *et* $\mathrm{I} = \Sigma_1 \times \Sigma_2$. *Les problèmes suivants sont indécidables :*

Donnée *Deux langages* $L_1,L_2 \in \mathrm{Rat}(\Sigma_1^* \times \Sigma_2^*) = \mathrm{Rat}(\mathbb{M}(\Sigma, \mathrm{D}))$.

Questions *A-t-on*

- $L_1 \cap L_2 = \varnothing$ *?*
- $L_1 \subseteq L_2$ *?*
- $L_1 = L_2$ *?*
- $L_1 = \Sigma^*$ *?*
- $\Sigma^* \setminus L_1$ *est fini ?*
- L_1 *est reconnaissable ?*

Tous ces problèmes sont décidables si L_1 *et* L_2 *sont des reconnaissables de* $\mathbb{M}(\Sigma,\mathrm{D})$.

Preuve. Ce théorème peut être raffiné suivant que la relation I est ou non transitive, [13]. Pour les langages reconnaissables, tous les problèmes sont trivialement décidables. Pour les langages rationnels, le problème de correspondance de Post (PCP) se réduit à chacun des problèmes. Voir [8] pour une preuve complète. On montre ici que le premier problème est indécidable.

Soit $\mathcal{I} = ((u_1,v_1),\ldots,(u_n,v_n))$ une instance de PCP. L'idée est de coder les concaténations de u_i (resp. de v_i) dans L_1 (resp. dans L_2). Une concaténation $u = u_{i_1} \cdots u_{i_k}$ se code par la suite $(i_1,\ldots,i_k)$ et le mot u. On code la suite sur Σ_1 et u sur Σ_2. Soient $a,b \in \Sigma_1$ et supposons (quitte à les coder sur deux lettres de Σ_2) que u_i,v_i sont écrits sur l'alphabet Σ_2. Ainsi, $L_1 = (ab\cdot u_1 + a^2b\cdot u_2 + \cdots + a^nb\cdot u_n)^*$ et $L_2 = (ab\cdot v_1 + a^2b\cdot v_2 + \cdots + a^nb\cdot v_n)^*$. Comme $\{a,b\} \mathrel{\mathrm{I}} \Sigma_2^*$, une trace de L_1, par exemple, est de la forme $a^{i_1}\cdots a^{i_k}\cdot u_{i_1}\cdots u_{i_k}$. Par définition, L_i est rationnel et $L_1 \cap L_2 \neq \varnothing$ ssi PCP a une solution positive sur l'instance $\mathcal{I}$. □

Théorème 3.1.2 *Soit* $L \subseteq \Sigma^*$ *rationnel reconnu par un automate* $\mathcal{A}$. *Une condition nécessaire pour que* $\bar{L}$ *soit reconnaissable est que toutes les boucles de* $\mathcal{A}$ *soient étiquetées par des traces connexes. Cette condition est testable sur* $\mathcal{A}$ *et* **co-NP**-*complète [31].*

3.2 Logiques temporelles sur les traces

La logique temporelle peut parler d'exécutions de systèmes concurrents. Les difficultés sont les suivantes : contrairement aux cas des systèmes séquentiels dont les comportements sont modélisés par des ordres totaux étiquetés (des mots), un sommet d'une trace peut comporter plusieurs successeurs. On peut donc introduire deux opérateurs next, l'un parlant de *tous* les sommets successeurs, l'autre parlant d'au moins un sommet successeur. De même, deux opérateurs Until sont utilisés, l'un existentiel, l'autre universel.

Par ailleurs, deux points de vue sont possibles. On peut considérer une logique qui parle d'états globaux du système, c'est-à-dire qui s'évalue sur les préfixes d'une trace. Ces logiques sont appelées *globales*. Au contraire, certaines logiques, dites *locales*, s'évaluent sur les sommets de la trace.

3.2.1 Logiques globales

Opérateurs globaux : syntaxe

Soit (Σ,D) un alphabets de dépendance. L'ensemble $\mathsf{LTL}(\Sigma,\mathrm{D}) = \mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,[a],\mathbf{EU},\mathbf{AU})$ $(a \in \Sigma)$ est le plus petit ensemble de formules contenant $\mathtt{tt}$, fermé par opérations Booléennes $\neg$ et $\vee$, et tel que

– si $\alpha \in \mathsf{LTL}(\Sigma,\mathrm{D})$, alors $\langle a\rangle\alpha$ et $[a]\alpha \in \mathsf{LTL}(\Sigma,\mathrm{D})$ pour tout $a \in \Sigma$;

– si $\alpha,\beta \in \mathsf{LTL}(\Sigma,\mathrm{D})$, alors $\alpha\,\mathbf{EU}\,\beta$ et $\alpha\,\mathbf{AU}\,\beta \in \mathsf{LTL}(\Sigma,\mathrm{D})$.

On utilise ici (sans confusion possible) les mêmes notations que pour la logique arborescente CTL^*. Si $\mathrm{Op} \subseteq \{\langle a\rangle,[a],\mathbf{EU},\mathbf{AU}\}$, on note $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\mathrm{Op})$ la logique n'utilisant que les opérateurs temporels de Op.

Opérateurs globaux : sémantique

Soit t une trace. On note $v \leqslant_p t$ si v est un préfixe de t et $v \lessdot_p t$ si $v \leqslant t$ et $v^{-1}t$ est un singleton (autrement dit, si t est obtenu de v en ajoutant un sommet maximal). On définit la relation de satisfaction $\models$ sur un couple t,v où $t \in \mathbb{M}(\Sigma,\mathrm{D})$ et $v \leqslant_p t$.

- $t,v \models \mathtt{tt}$ ($\mathtt{tt}$ représente *vrai*) ;
- $t,v \models \alpha \vee \beta$ si $t,v \models \alpha$ ou $t,v \models \beta$;
- $t,v \models \neg\alpha$ si l'on n'a pas $t,v \models \alpha$;
- $t,v \models \langle a\rangle\alpha$ si $va \leqslant_p t$ et $t,va \models \alpha$;
- $t,v \models [a]\alpha$ si $va \leqslant_p t \Rightarrow t,va \models \alpha$;
- $t,v \models \alpha\,\mathbf{EU}\,\beta$ si soit $t,v \models \beta$, soit il existe $v = v_1 \lessdot_p \cdots \lessdot_p v_n \lessdot_p w$ tels que
  – $t,v_i \models \alpha$ pour $i = 1,\ldots,n$;
  – $t,w \models \beta$,
- $t,v \models \alpha\,\mathbf{AU}\,\beta$ s'il existe w, $v \leqslant_p w \leqslant_p t$ tel que
  – $\forall u \leqslant_p t,\ v \leqslant_p u <_p w \Rightarrow t,u \models \alpha$;
  – $t,w \models \beta$,

On définit comme d'habitude $L(\alpha) = \{t \in \mathbb{M}(\Sigma,\mathrm{D}) \mid t,1 \models \alpha\}$ (où 1 désigne la trace vide). On voit facilement que $[a]\alpha = \neg\langle a\rangle\neg\alpha$, on peut donc ne considérer qu'un seul de ces opérateurs. Par contre, $\mathbf{EU}$ et $\mathbf{AU}$ ne sont pas duaux (on peut introduire les opérateurs *release* existentiels et universels). On peut aussi introduire les équivalents du *next* sur les mots $\mathbf{EX}\alpha = \bigvee_{a\in\Sigma}\langle a\rangle\alpha$ et $\mathbf{AX}\alpha = \neg\mathbf{EX}\neg\alpha$. Le théorème suivant résume quelques propriétés importantes des logiques globales. Certains de ces résultats se généralisent aux comportements infinis.

Théorème 3.2.1 *(i) Le* model-checking *et la satisfaisabilité de* $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,\mathbf{EU})$ *sont indécidables.*

(ii) La satisfaisabilité de $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,\mathbf{AU})$ *est non élémentaire.*

(iii) On peut effectivement construire un automate reconnaissant toutes les linéarisations d'une formule de $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,\mathbf{AU})$*.*

Preuve. On prouve ici seulement (i), résultat du à R. Alur et D. Peled [4]. On fait ici une réduction différente, inspirée de la preuve de I. Walukiewicz [41] pour (ii). Le résultat (iii) est du à P. Gastin, R. Meyer et A. Petit [16]. Pour (i), on code le calcul d'une machine de Turing M sur le mot vide par une formule $\varphi(M)$ de $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,\mathbf{EU})$. On montre que $L(\varphi(M)) = \varnothing$ ssi M n'a pas de calcul acceptant sur le mot vide, d'où l'indécidabilité de la satisfaisabilité de $\mathsf{LTL}_{\Sigma,\mathrm{D}}(\langle a\rangle,\mathbf{EU})$. Le même résultat pour le model-checking en découle facilement, il suffit de vérifier $\varphi(M)$ sur un automate acceptant toutes les (linéarisations de) traces.

Soit donc M une machine de Turing d'alphabet de travail Γ, où $\square \in \Gamma$ est le symbole blanc. Les configurations de M sont des mots de $(\Gamma\setminus Q)^*Q(\Gamma\setminus Q)^*$ où Q est l'ensemble des états de M. La configuration uqv indique que uv est écrit sur la bande, que M est dans l'état q et que la tête de lecture est sur la première lettre de v. On code une configuration de M sur un alphabet de la forme (Σ,D), avec $\Sigma = \Sigma_1 \cup \bar{\Sigma}_1$, où $\bar{\Sigma}_1$ est une copie (disjointe) de Σ_1 et où $\mathrm{D} = (\Sigma_1 \times \Sigma_1) \cup (\bar{\Sigma}_1 \times \bar{\Sigma}_1)$. On prend $\Sigma_1 = \big(\Gamma \times \{0,1,2\}\big) \cup \{\$,\#\}$.

On peut toujours supposer qu'une configuration $C = c_1 \cdots c_k$ de M est de taille k divisible par 3 (quitte à inclure un ou deux symboles blancs supplémentaires à la fin). On note $i_{[3]}$ la valeur de i modulo 3 dans $\{1,2,3\}$. On représente la configuration C par la trace $t(C) = [\#(c_1,1_{[3]})\cdots(c_k,k_{[3]})][\bar{\#}\,\overline{(c_1,1_{[3]})}\cdots\overline{(c_k,k_{[3]})}]$ de $\mathbb{M}(\Sigma,\mathrm{D})$. On représente un calcul $C_0 \vdash C_1 \vdash \cdots \vdash C_n$ de la machine M par $t(C_0)t(C_1)\cdots t(C_n)\$\bar{\$}$.

Il reste à écrire la formule $\varphi(M)$ acceptant les traces qui représentent un calcul acceptant de M. Pour $i = 0,1,2$, on pose (abusivement) $\langle i\rangle = \bigvee_{x\in\Gamma}\langle(x,i)\rangle\mathsf{tt}$ et $\langle\bar{i}\rangle = \bigvee_{x\in\Gamma}\langle\overline{(x,i)}\rangle\mathsf{tt}$. Finalement, si A est un ensemble de lettres, on pose $\langle A\rangle\varphi = \bigvee_{a\in A}\langle a\rangle\varphi$. On définit ensuite les formules suivantes.

$$\alpha_= \equiv (\langle\#\rangle\langle\bar{\#}\rangle\mathsf{tt}) \vee \bigvee_{i=1,2,3} \langle i\rangle\langle\bar{i}\rangle\mathsf{tt}$$

$$\alpha_{\mathrm{succ}} \equiv (\langle 1\rangle\langle\bar{\#}\rangle\mathsf{tt}) \vee (\langle\#\rangle\langle\bar{3}\rangle\mathsf{tt}) \vee \bigvee_{i=1,2,3} \langle i+1\rangle\langle\bar{i}\rangle\mathsf{tt}$$

$$\beta_{\mathrm{succ}} \equiv (\langle\$\rangle\langle\bar{3}\rangle\mathsf{tt}) \vee \alpha_{\mathrm{succ}}$$

$$\alpha_{\mathrm{same}} \equiv \bigvee_{i=1,2,3}\bigvee_{x\in\Gamma}\langle(x,i)\rangle\langle\overline{(x,i)}\rangle\mathsf{tt}$$

La formule $\alpha_=$ est vraie sur les configurations qui voient soit le même indice sur Σ_1 et $\bar{\Sigma}_1$, soit un $\#$ et un $\bar{\#}$. Les formules α_{succ} et β_{succ} traduisent le fait que la configuration voit une lettre non barrée « en avance d'un cran » sur la lettre barrée (on y interprète $i+1$ modulo 3 dans $\{0,1,2\}$). Finalement α_{same} est vraie dans des configurations qui voient sur $\Gamma,\bar{\Gamma}$ deux lettres identiques de même indice. Il est facile d'écrire une formule φ_{init} qui accepte les traces commençant par la configuration initiale de M. Il est également facile d'écrire une formule φ_{fin} assurant que la configuration finale est atteinte avant d'atteindre $\$\bar{\$}$. Soit q_0 est l'état de départ de M, et supposons que M accepte sur bande vide dans q_f.

$$\begin{aligned}
\varphi_{\text{init}} &\equiv \langle\#\rangle\langle(q_0,1)\rangle\langle(\square,2)\rangle\langle(\square,3)\rangle\langle\#\rangle\langle\bar{\#}\rangle\langle\overline{(q_0,1)}\rangle\langle\overline{(\square,2)}\rangle\langle\overline{(\square,3)}\rangle\langle\bar{\#}\rangle\mathsf{tt}\\
\varphi_{\text{fin}} &\equiv \mathsf{tt}\,\mathbf{EU}\,\alpha_{\text{fin}}\\
\alpha_{\text{fin}} &\equiv \langle\$\rangle\langle\bar{\$}\rangle \bigwedge_{a\in\Sigma_1}(\neg\langle a\rangle\mathsf{tt}\wedge\neg\langle\bar{a}\rangle\mathsf{tt})
\end{aligned}$$

On doit maintenant exprimer que les projections de $t(C_0)\cdots t(C_n)$ ont la même forme sur Σ_1 et $\bar{\Sigma}_1$.

$$\begin{aligned}
\varphi_{\text{same_shape}} &\equiv \alpha_{\text{same_shape}}\,\mathbf{EU}\,\alpha_{\text{fin}}\\
\alpha_{\text{same_shape}} &\equiv \big(\alpha_{=} \Rightarrow (\langle\#\rangle\langle\bar{\#}\rangle\mathsf{tt}\vee\alpha_{\text{same}})\wedge\langle\Sigma_1\rangle(\beta_{\text{succ}}\wedge\langle\bar{\Sigma}_1\rangle(\langle\$\rangle\langle\bar{\$}\rangle\mathsf{tt}\vee\alpha_{=})\big)
\end{aligned}$$

Pour exprimer enfin qu'une trace représente un calcul, il faut assurer qu'entre les $i^{\text{ème}}$ et $(i+1)^{\text{ème}}$ $\#$ se trouve le codage d'une configuration C_i et que $C_i \vdash C_{i+1}$.

$$\begin{aligned}
\varphi_{\text{step}} &\equiv \alpha_{\text{step}}\,\mathbf{EU}\,\alpha_{\text{fin}}\\
\alpha_{\text{step}} &\equiv \alpha_{\text{same_shape}}\wedge\Big(\langle\#\rangle\langle\bar{\#}\rangle\mathsf{tt}\Rightarrow\langle\#\rangle\big[(\langle\bar{\#}\rangle\mathsf{tt}\wedge\langle\Gamma\rangle\mathsf{tt})\,\mathbf{EU}\,\langle\#\rangle\langle\bar{\#}\rangle\beta_{\text{step}}\big]\Big)
\end{aligned}$$

Finalement, β_{step} doit accepter les configurations commençant par une configuration C_{i+1} sur Σ_1 et $\bar{C}_i$ sur $\bar{\Sigma}_i$. La formule est du même type que $\varphi_{\text{same\_shape}}$, sauf qu'elle attend $\langle\#\rangle\langle\bar{\#}\rangle$ ou $\langle\$\rangle\langle\bar{\#}\rangle$ en fin d'*until*, et teste s'il y a une transition (ce qui se fait en regardant 3 positions consécutives sur Σ_1 et $\bar{\Sigma}_1$ commençant au même indice). Elle n'est pas difficile à écrire mais on s'en dispense facilement (l'écriture est rendue pénible par le fait que si la bande grandit, il faut ajouter des blancs). □

3.2.2 Logiques locales

Contrairement aux logiques globales, les formules des logiques locales s'évaluent en un sommet d'une trace et non en une configuration. L'intérêt de ces logiques est que les problèmes de satisfaisabilité et de *model-checking* associés sont décidables et dans PSPACE (pas plus coûteux en complexité que pour LTL). La logique présentée ici a été introduite dans [12] par V. Diekert et P. Gastin. Il existe de nombreuses variantes de telles logiques, voir par exemple [21]. Une logique locale de la même expressivité que la logique du premier ordre a été proposée récemment par P. Gastin et M. Mukund [17].

Opérateurs locaux : syntaxe

Pour définir les formules locales, on a besoin, de façon intermédiaire, de formules *internes*. L'ensemble $\mathsf{LocTL}^i_{\Sigma,\mathrm{D}}$ de ces formules est le plus petit ensemble de formules contenant tt et les lettres de l'alphabet Σ, fermé par opérations Booléennes $\neg$ et $\vee$, et tel que

– si $\alpha\in\mathsf{LocTL}^i_{\Sigma,\mathrm{D}}$, alors $\mathbf{EX}\alpha\in\mathsf{LocTL}^i_{\Sigma,\mathrm{D}}$;

– si $\alpha,\beta \in \mathsf{LTL}(\Sigma,\mathrm{D})$, alors $\alpha\,\mathbf{EU}\,\beta$ et $\alpha\,\mathbf{AU}\,\beta \in \mathsf{LTL}(\Sigma,\mathrm{D})$.

On utilise ici sans confusion possible les mêmes notations que pour certaines logiques précédemment définies. Si $\mathrm{Op} \subseteq \{\mathbf{EX},\mathbf{EU},\mathbf{AU}\}$, on note $\mathsf{LocTL}_{\Sigma,\mathrm{D}}(\mathrm{Op})$ la logique n'utilisant que les opérateurs temporels de Op. Comme d'habitude, on peut définir **AX** comme le dual de **EX**, et introduire les opérateurs *release* **AR** et **ER**, duaux de **EU** et **AU**, respectivement.

Une formule locale de $\mathsf{LocTL}_{\Sigma,\mathrm{D}}$ est par définition une combinaison Booléenne de formules de la forme $\mathbf{EM}\varphi$, où φ est une formule de $\mathsf{LocTL}^{i}_{\Sigma,\mathrm{D}}$.

Opérateurs locaux : sémantique

Soit $t = (V,\leqslant,\lambda)$ une trace et $\varphi \in \mathsf{LocTL}^{i}_{\Sigma,\mathrm{D}}$. On définit la relation de satisfaction $\models$ sur t,x où $x \in V$:

- $t,x \models \mathtt{tt}$ (tt représente *vrai*) ;
- $t,x \models a$ si $\lambda(x) = a$;
- $t,x \models \alpha \vee \beta$ si $t,x \models \alpha$ ou $t,x \models \beta$;
- $t,x \models \neg\alpha$ si l'on n'a pas $t,x \models \alpha$;
- $t,x \models \mathbf{EX}\alpha$ s'il existe un sommet $y \in V$ tel que $x \lessdot y$ et $t,y \models \alpha$;
- $t,x \models \alpha\mathbf{EU}\beta$ si soit $t,x \models \beta$, soit il existe une suite de sommets $x = x_1 \lessdot \cdots \lessdot x_n \lessdot y$ tels que
  - $t,x_i \models \alpha$ pour $i = 1,\ldots,n$;
  - $t,y \models \beta$,
- $t,x \models \alpha\,\mathbf{AU}\,\beta$ s'il existe $y \in V$, $x \leqslant y$ tel que
  - $\forall z \in V$, $x \leqslant z < y \Rightarrow t,z \models \alpha$;
  - $t,y \models \beta$,

Les modèles des formules de $\mathsf{LocTL}_{\Sigma,\mathrm{D}}$ sont des traces de $\mathbb{M}(\Sigma,\mathrm{D})$. Les opérateurs Booléens ont leur sens habituel, et, si $t = (V, \leqslant, \lambda) \in \mathbb{M}(\Sigma,\mathrm{D})$, on a :

– ∘ $t\,\mathbf{EM}\,\alpha$ s'il existe un sommet x de V, minimal pour $\leqslant$, tel que $t,x \models \alpha$.

Pour $\alpha \in \mathsf{LocTL}_{\Sigma}$, on définit comme d'habitude $L(\alpha) = \{t \in \mathbb{M}(\Sigma,\mathrm{D}) \mid t \models \alpha\}$. Le théorème suivant résume deux propriétés importantes de cette logique. Certains de ces résultats se généralisent aux comportements infinis.

Théorème 3.2.2 (*i*) *Le* model-checking *et la satisfaisabilité de* $\mathsf{LocTL}_{\Sigma,\mathrm{D}}(\mathbf{EX},\mathbf{EU},\mathbf{AU})$ *sont* ***PSPACE****(et donc* ***PSPACE****-complets, comme les problèmes correspondants sur* LTL*).*

(*ii*) *Les logiques* $\mathsf{LocTL}_{\Sigma,\mathrm{D}}(\mathbf{EX},\mathbf{EU})$ *et* $\mathsf{LocTL}_{\Sigma,\mathrm{D}}(\mathbf{EX},\mathbf{AU})$ *sont expressivement complètes (*ie*, ont même pouvoir d'expression que la logique du premier ordre) si et seulement si l'alphabet de dépendance est un cographe.*

Pour la preuve de ce théorème, voir [12]. On remarquera que **AU** est toujours exprimable au premier ordre, alors que, pour certains alphabets de dépendance (qui ne sont pas des cographes), on peut exprimer des propriétés qui ne sont pas du premier ordre en $\mathsf{LocTL}_{\Sigma,\mathrm{D}}(\mathbf{EX},\mathbf{EU})$.

3.3 Message Sequence Charts et Automates Communicants

Les diagrammes de séquence (Message Sequence Charts, ou MSCs) sont un formalisme permettant de décrire des protocoles de communication, normalisé dans une recommandation de l'ITU (International Telecommunication Union [1]). Ils permettent de spécifier des propriétés de systèmes de processus qui communiquent via des canaux point-à-point.

Définition 3.3.1 (MSC) *Un MSC sur un ensemble $\mathcal{P}$ est un n-uplet $M = \langle E\,, <\,, \mathcal{P}\,, t\,, \mathcal{C}\,, f\rangle$ où :*

- *$\mathcal{P}$ est un ensemble fini de* processus *(ou* instances*).*
- *$E = \biguplus_{p\in\mathcal{P}} E_p$ est l'union disjointe des ensembles E_p. Les éléments de E_p sont appelés* événements *(ou* actions*) du processus p. Pour $e \in E$, on note $P(e) \in \mathcal{P}$ l'unique processus p tel que $e \in E_p$, et on dit que e est un* événement *du processus p.*
- *Chaque événement est soit un événement de communication (envoi ou réception) ou un événement local. On écrit $E = S \uplus R \uplus L$ comme une union disjointe, avec S l'ensemble des envois, R l'ensemble des réceptions et L l'ensemble des événements locaux.*
- *$\mathcal{C}$ est un ensemble fini de contenus (ie, de noms) de messages ou d' action locales.*
- *$t : E \to A = \{p!q(a), p?q(a), l_p(a) \mid p,q \in \mathcal{P},\ p \neq q,\ a \in \mathcal{C}\}$ étiquette chaque événement par son* type *$t(e)$, avec*
  - *$t(e) = p!q(a)$ si $e \in E_p \cap S$ est un envoi du message a de p à q,*
  - *$t(e) = p?q(a)$ si $e \in E_p \cap R$ est une réception du message a par p de q*
  - *$t(e) = l_p(a)$ si $e \in E_p \cap L$ est un événement local de p nommé a.*
- *$f : S \to R$ est une bijection qui associe chaque envoi avec sa réception. Si $f(e) = e'$, alors il existe $p,q \in \mathcal{P}$ et $a \in \mathcal{C}$ tels que $t(e) = p!q(a)$ et $t(e') = q?p(a)$.*
- *Pour chaque $p \in \mathcal{P}$, E_p est muni d'une relation d'ordre <u>total</u> $<_p$: tous les événements de p sont totalement ordonnés (dans le temps). La relation $< \subseteq E \times E$ est la plus petite relation telle que :*
  - *La restriction de $<$ à E_p est $<_p$, pour chaque $p \in \mathcal{P}$.*
  - *Pour tous $e,f \in E$, $m(e) = f$ implique $e < f$.*

  On demande que cette relation soit acyclique (et donc qu'elle induise un ordre partiel sur E). Elle est appelée ordre visuel *du MSC.*
  L'ensemble de tous les MSCs finis est noté $\mathbb{bMSC}$.

Les MSCs sont utilisés pour décrire des protocoles de communication entre processus. Ceux-ci sont supposés communiquer via des buffers. Chaque paire (p,q) de processus dispose d'un buffer $B_{p,q}$ de p vers q et d'un buffer $B_{p,q}$ de p vers q. Dans ce cours, on suppose sauf mention contraire que ces buffers FIFO et fiables. La figure 3.4 représente quatre MSC. Les messages entre eux sont représentés par des flèches horizontales (le nom des messages n'est pas mentionné). Les événements d'envoi sont situés au départ des flèches et les événements de réception à l'arrivée. Il n'y a pas d'événements locaux sur ces exemples. Le temps s'écoule de haut en bas (*ie*, les ordres E_p sont représentés sur chaque ligne verticale de haut en bas). Les trois premiers MSCs (a), (b), (c) sont FIFO, le quatrième (d) ne l'est pas. La relation $<$ associée au dernier graphe (e) est cyclique, ce graphe ne représente pas un MSC.

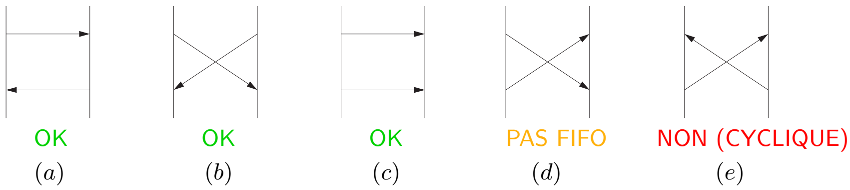


FIG. 3.4 – *MSCs : exemples*

Un autre exemple est donné Figure 3.5. Les processus sont un Client, un Serveur et une Imprimante. Les étiquettes des messages (en rouge) sont leurs noms, et les pointillés montrent que l'envoi de Job_2 par le processus Client précède l'envoi de Fail_1 par le processus Imprimante, dans l'ordre $<$.

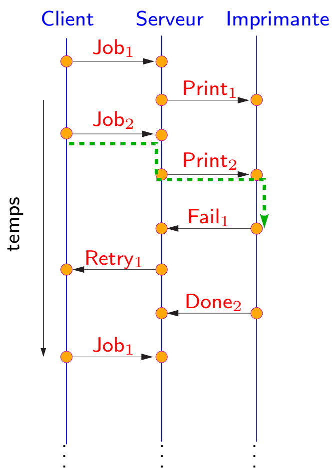


FIG. 3.5 – *Un exemple de MSC*

Le modèle de machine le plus courant pour cette architecture est celui d'automate communicant (Communicating Finite state Machine, ou CFM).

Définition 3.3.2 (CFM) *Un CFM $\mathcal{A}$ est un ensemble d'automates finis $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ qui communiquent via des buffers non bornés et fiables. On supposera, sauf mention contraire, que les buffers sont FIFO. À chaque couple $(p,q) \in \mathcal{P}^2$ de processus distincts, on associe un buffer $B_{p,q}$. Chaque (contenu de) buffer est représenté par un mot sur un alphabet fini $\mathcal{C}$. Chaque automate $\mathcal{A}_p$ est décrit par un n-uplet $\mathcal{A}_p = (S_p, A_p, \rightarrow_p, F_p)$ consistant en un ensemble d'états locaux S_p, un ensemble d'actions A_p, un ensemble d'états finals locaux F_p et une relation de transition $\rightarrow_p \subseteq S_p \times A_p \times S_p$: les étiquettes des transitions de $\mathcal{A}_p$ sont des actions. Le calcul de $\mathcal{A}$ commence dans un état initial $s^0 \in \prod_{p\in\mathcal{P}} S_p$. Les actions de $\mathcal{A}_p$ sont soit des actions locales soit des envois/réceptions de message. On utilise les mêmes notations que pour les MSCs. L'envoi d'un message $a \in \mathcal{C}$ de p à q est noté $p!q(a)$, et si $\mathcal{A}_p$ exécute une transition avec cette étiquette, alors la lettre a est ajoutée* *en fin de buffer $B_{p,q}$. La réception du message a par p depuis q est notée $p?q(a)$; une transition étiquetée par cette action ne peut avoir lieu que si a est le premier message (*ie*, la première lettre) du buffer (FIFO) $B_{q,p}$, et l'effet de la transition est alors de supprimer la première occurrence de a du buffer $B_{q,p}$.*

Une configuration *$C = (q,B)$ d'un CFM $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ est donc décrite par un état global $\prod_{p\in\mathcal{P}} S_p$ et les contenus des buffers $B \in (\mathcal{C}^*)^{\mathcal{P}\times\mathcal{P}}$. La relation de transition du CFM est notée $\rightarrow$, sa clôture transitive et réflexive est notée par $\xrightarrow{*}$. Formellement, les transitions de $\mathcal{A}$ sont les suivantes.*

$$((q_r)_{r\in\mathcal{P}},(B_{r,s})_{r\neq s}) \xrightarrow{p!q(a)} ((q'_r)_{p\in\mathcal{P}},(B'_{r,s})_{r\neq s}) \quad si \begin{cases} q_r = q'_r & si\ r \neq p \\ q_p \rightarrow_p q'_p & \\ B'_{r,s} = B_{r,s} & si\ (r,s) \neq (p,q) \\ B'_{p,q} = B_{p,q} \cdot a & \end{cases}$$

$$((q_r)_{r\in\mathcal{P}},(B_{r,s})_{r\neq s}) \xrightarrow{p?q(a)} ((q'_r)_{p\in\mathcal{P}},(B'_{r,s})_{r\neq s}) \quad si \begin{cases} q_r = q'_r & si\ r \neq p \\ q_p \rightarrow_p q'_p & \\ B'_{r,s} = B_{r,s} & si\ (r,s) \neq (p,q) \\ B'_{p,q} = a^{-1}B_{p,q} & \end{cases}$$

La configuration d'état global s^0 avec des buffers vides est la configuration initiale. Une exécution *$\sigma = C_1 \xrightarrow{a_1} C_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{m-1}} C_m$ de $\mathcal{A}$ est un $\rightarrow$-chemin fini. L'étiquetage de l'exécution σ est la suite $a_1 \cdots a_{m-1}$. L'étiquetage d'une exécution σ définit naturellement un MSC partiel $msc(\sigma)$ (c'est-à-dire, le préfixe d'un MSC, au sens donné ci-dessous).*

Une exécution est acceptante *si chaque processus p finit dans un état final de F_p et tous les buffers sont vides. L'ensemble des exécutions acceptantes de $\mathcal{A}$ est noté $\mathcal{L}(\mathcal{A})$ et est un ensemble de MSCs finis.*

Théorème 3.3.1 *Étant donné un CFM on ne peut pas décider si un message particulier sera envoyé et reçu, ou si une configuration particulière est atteignable, ou si une configuration est un blocage (*ie*, aucun run acceptant ne part de cette configuration).*

Idée de preuve. Provient du fait qu'on peut simuler le calcul d'une machine de Turing sur un canal entre deux automates locaux. □

La spécification d'un protocole de communication faisant apparaître de nombreux scénarios, on a besoin d'une description de haut niveau pour décrire comment ils peuvent s'enchaîner, et définir ainsi des ensembles infinis de comportements. La description donnée dans le standard de la norme Z.120 de l'ITU [1] utilise le choix non déterministe, la concaténation et l'itération pour définit des ensembles infinis de MSCs.

Pour cela, on a besoin de définir un produit de MSCs. On utilise ici le produit *asynchrone*. De façon informelle, pour concaténer M_1 et M_2, on colle M_2 sous M_1 : soient $M_1 = \langle E_1, <_1, \mathcal{P}, t_1, \mathcal{C}_1, f_1\rangle$ et $M_2 = \langle E_2, <_2, \mathcal{P}, t_2, \mathcal{C}_2, f_2\rangle$ deux MSCs sur le même ensemble de processus $\mathcal{P}$ (ce qu'on peut toujours supposer, quitte à prendre l'union des ensembles de processus de M_1 et de M_2). Le *produit* M_1M_2 est le MSC $\langle E_1 \cup E_2, <, \mathcal{P}, t_1 \cup$

$t_2, \mathcal{C}_1 \cup \mathcal{C}_2, f_1 \cup f_2\rangle$ sur l'union disjointe des événements $E_1 \uplus E_2$, avec l'ordre visuel donné par :

$$< \; = \; <_1 \cup <_2 \cup \{(e,f) \in E_1 \times E_2 \mid P(e) = P(f)\}.$$

Ainsi, les événements de M_1 précédent ceux de M_2 pour chaque processus, respectivement. Par contre, il n'y a pas de synchronisation entre différents processus. Il est possible qu'un processus soit toujours dans M_1, tandis qu'un autre a commencé ses actions dans M_2. On dit que M_1 est un *préfixe* de M_1M_2.

Exemple 3.3.1 *Soit M_0 le MSC sur deux processus p,q suivant.*

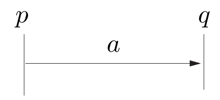


FIG. 3.6 – *Le MSC M_0*

On constate qu'il existe des linéarisations de M_0^n dans lesquelles p exécute tous *ses événements sans que q n'en ait exécuté un seul.*

Définition 3.3.3 (HMSC) *Un* High-Level MSC *(HMSC) G est un automate fini dont les arêtes sont étiquetées par des MSCs finis. L'étiquette d'un chemin $q_0 \xrightarrow{M_1} q_1 \xrightarrow{M_2} \cdots \xrightarrow{M_{k-1}} q_{k-1} \xrightarrow{M_k} q_k$ est le MSC $M_1 \cdots M_k$. Le langage $\mathcal{L}(G)$ défini par G est l'ensemble des étiquettes des chemins acceptants. Les éléments de $\mathcal{L}(G)$ sont appelés* exécutions *de G. Se donner un HMSC est donc équivalent à se donner une expression rationnelle sur l'alphabet $\mathbb{bMSC}$.*

Étant donné un MSC M dont l'ensemble d'événements est $\{e_1, \ldots, e_k\}$, l'ensemble des linéarisations de M est

$$\mathsf{Lin}(M) = \{t(e_1\} \cdots t(e_k) \mid e_i < e_j \Rightarrow i < j\}$$

Si M est FIFO, on peut reconstruire M à partir de l'une quelconque de ses linéarisations. Étant donné un langage de MSCs $L \subseteq \mathbb{bMSC}$, l'ensemble des linéarisations de L est $\mathsf{Lin}(L) = \bigcup_{M \in L} \mathsf{Lin}(M)$. L'ensemble des linéarisations des MSCs acceptés par un HMSC G est noté $\mathsf{Lin}(G)$.

Remarque L'ensemble $\mathsf{Lin}(G)$ n'est pas en général rationnel. Il suffit de considérer M^* où M est le MSC de l'exemple 3.3.1.

Un *atome* est un MSC qui n'est pas produit non trivial de deux MSCs. L'ensemble des atomes est noté $\mathbb{At}$.

$$M \in \mathbb{At} \text{ ssi } (M = M_1M_2 \Longrightarrow M = M_1 \text{ ou } M = M_2)$$

Par $P(M)$, on désigne les processus actifs dans le MSC M (*ie*, ceux qui exécutent au moins une action). On considère la relation suivante de dépendance sur $\mathbb{bMSC}$: $M_1 \mathrm{D} M_2$ ssi $P(M_1) \cap P(M_2) \neq \varnothing$. Il est facile de voir que $\mathbb{M}(\mathbb{At}, \mathrm{D})$ est un monoïde de traces. Il n'est pas engendré par un nombre fini d'éléments car on peut construire un nombre infini d'atomes, même sur un alphabet de messages à une lettre et sur deux processus.

Il est aussi facile de voir que tout MSC se décompose en un produit d'atomes et que cette décomposition est unique à permutation près d'atomes indépendants. Ainsi, $\mathbb{bMSC}$est (isomorphe à) un monoïde de traces, ce qui explique beaucoup des résultats d'indécidabilité des sections suivantes. Beaucoup de ces résultats d'indécidabilité sont présentés dans [31, 30, 32],.

3.3.1 Classes de HMSCs

Les dernières remarques ont conduit à considérer des sous-classes de HMSCs, pour lesquelles les problèmes habituels sont décidables.

HMSCs à canaux bornés

Étant donné un préfixe N d'un MSC M et $p,q \in \mathcal{P}$, $p \neq q$, le nombre d'envois (resp. de réceptions) de messages de p à q (resp. de q depuis p) dans N est noté $|N|_{p!q}$ (resp. $|N|_{q?p}$). Le nombre de messages pendants dans le buffer $B_{p,q}$ après exécution de N est donc $\wp_{(p,q)}(N) = |N|_{p!q} - |N|_{q?p}$.

Un HMSC G est dit *à canaux bornés* s'il existe une borne $b \in \mathbb{N}$ telle que

$$\forall M \in \mathcal{L}(G), \forall p,q \in \mathcal{P},\ p \neq q,\ \forall N \text{ préfixe de } M, \text{ on a } \wp_{(p,q)}(N) \leqslant b$$

La terminologie des HMSCs étant particulièrement bien faite, il ne faut pas confondre dans la littérature HMSCs à canaux bornés (*channel bounded*) et HMSCs bornés (*bounded*).

HMSCs réguliers

Un HMSC G est dit *régulier* si le langage (de mots) $\mathsf{Lin}(G)$ est rationnel. Les HMSCs réguliers ont été étudiés dans [20, 19]. Les problèmes de décision comme le *model-checking* (cf. ci-dessous) sont bien entendu décidables pour cette classe de HMSCs.

HMSCs reconnaissables

Comme l'ensemble des MSCs finis est isomorphe au monoïde de traces $\mathbb{At}$, D, on peut définir comme d'habitude la notion de langage de MSCs reconnaissables. Un langage de MSCs L est *reconnaissable* s'il existe un morphisme η de $\mathbb{bMSC}$ dans monoïde fini tel que $L = \eta^{-1}(\eta(L))$. De façon équivalente, L est reconnaissable si les ensembles $LM^{-1}(M \in \mathbb{bMSC})$ sont en nombre finis. Un HMSC G est dit reconnaissable si $\mathcal{L}(G)$ est reconnaissable.

Exemple 3.3.2 *Soit M_0 le MSC de l'exemple 3.3.1. Le HMSC donné par l'expression rationnelle M_0^* est reconnaissable, mais non régulier. Il n'est pas non plus à canaux bornés.*

Soit M_1 le MSC suivant.

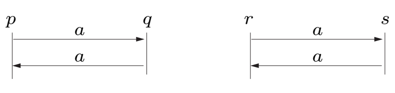


FIG. 3.7 – *Le MSC M_1*

On vérifie que le HMSC donné par l'expression rationnelle M_q^* est à canaux bornés, mais ni reconnaissable (même argument que pour $(ab)^*$, $a \mathrm{I} b$), ni régulier.

Graphe de communication

Définition 3.3.4 *Le* graphe de communication $gr(M)$ *d'un MSC M est un graphe dirigé défini de la façon suivante. Les sommets du graphe sont les processus, et on a une arête $p \to q$ si et seulement si p envoie un message à q dans M.*

Un MSC est connexe si son graphe de communication est faiblement connexe.

Un HMSC est localement coopératif si toutes ses étiquettes sont des MSCs connexes et pour tout chemin $s_1 \xrightarrow{M_1} s_2 \xrightarrow{M_2} s_3$, le MSC M_1M_2 est connexe.

Un HMSC est globalement coopératif si toutes ses étiquettes sont des MSCs connexes et pour toute boucle $s_1 \xrightarrow{M_1} s_2 \xrightarrow{M_2} \cdots \xrightarrow{M_n} s_1$, le MSC $M_1M_2 \cdots M_n$ est connexe. On peut remarquer que, quitte à remplacer un MSC par sa décomposition en MSCs atomiques, on peut supposer que les étiquettes sont atomiques.

On peut vérifier facilement le résultat suivant, dû à R. Morin [28].

Proposition 3.3.2 *Un HMSC G est à canaux bornés si et seulement si le graphe de communication de chaque MSC étiquetant une de ses boucles simples a toutes ses composantes connexes fortement connexes. En particulier, on peut décider si un HMSC est à canaux bornés.*

Le théorème suivant résulte facilement des résultats d'indécidabilité sur les traces.

Proposition 3.3.3 *Savoir si un HMSC est reconnaissable est indécidable.*

Théorème 3.3.4 *Un HMSC est régulier si et seulement s'il est reconnaissable et à canaux bornés.*

Voir [28] pour une preuve. On déduit immédiatement des trois résultats précédents la proposition suivante.

Proposition 3.3.5 *On ne peut pas décider si un HMSC est régulier.*

On peut relier graphe de communication des boucles et la régularité ou la reconnaissabilité d'un HMSC [31, 20].

Proposition 3.3.6 *Un HMSC est régulier (resp. reconnaissable) si et seulement s'il existe un autre HMSC reconnaissant le même langage dont toutes les boucles sont étiquetées par des MSCs dont le graphe de communication est fortement connexe (resp. connexe).*

3.3.2 Model-checking de HMSCs

On peut utiliser les HMSCs à la fois pour modéliser les protocoles et pour spécifier des propriétés à vérifier. Dans ce contexte étant donnés deux HMSCs G,H, savoir si l'on a $\mathcal{L}(G) \cap \mathcal{L}(H) = \varnothing$ ou $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ sont des problèmes de model-checking. Le cas du model-checking négatif (le plus facile) est déjà indécidable.

Théorème 3.3.7 *Savoir si deux HMSCs ont une exécution commune est indécidable.*

Idée de preuve. On réduit le problème de correspondance de Post à se problème. L'idée de la preuve est de construire, à partir d'une instance $\{(u_i,v_i)\}_i$de PCP, un HMSC qui génère le codage de concaténations de (u_i,v_i) et un HMSC qui teste que ces concaténations sont égales. Le codage peut se faire sur quatre processus, p_1,p_2,p_3,p_4. Dans le MSC M_i codant la paire (u_i,v_i), p_1 envoie à p_2 les lettres de u_i, qui sont reçues dans l'ordre et p_3 envoie à p_4 celles de v_i, reçues également dans l'ordre. □

Pour les HMSCs réguliers, les problèmes de model-checking sont trivialement décidables. La plus grande classe de HMSCs pour laquelle le model-checking est connu comme étant décidable est la classe des HMSCs globalement coopératifs [18].

Théorème 3.3.8 – *Savoir si deux HMSCs globalement coopératifs ont une exécution commune est PSPACE-complet.*
- *Décider de l'inclusion des ensembles d'exécutions de deux HMSCs globalement coopératifs est EXPSPACE-complet.*
- *Si on fixe le nombre de processus, la complexité des problèmes de model-checking pour les HMSCs localement coopératifs est NLOGSPACE et PSPACE-complets.*

3.3.3 Réalisation (ou implémentation) de HMSCs

Un autre problème est de savoir si, étant donné un HMSC, on peut en *distribuer* le contrôle, c'est à dire, si on peut trouver un CFM qui accepte *exactement* le même langage. Ce problème s'appelle implémentabilité (ou réalisabilité) du HMSC. Cette notion a été introduite dans [2] et étudiée pour des langages finis de HMSCs.

Une implémentation est *sûre* si elle n'atteint jamais de configuration de blocage.

En général, le problème de distribution de contrôle est difficile. Pour les langages reconnaissables de traces, ce problème a été résolu par l'élégante construction de W. Zielonka [42].

Il est facile de constater que chaque processus p d'un CFM qui accepte le même langage qu'un HMSC G doit être « implémenté » par un automate $\mathcal{A}_p$ qui reconnaît exactement la projection de $\mathcal{L}(G)$ sur p.

À nouveau, on montre l'indécidabilité de ce problème, même dans le cas régulier [3, 24].

Théorème 3.3.9 *Le problème de savoir si un HMSC régulier est implémentable par un CFM FIFO est indécidable. Le problème devient EXPSPACE-complet si on recherche une implémentation sûre.*

Le résultat ci-dessus contraste avec celui de R. Morin [29] qui prouve que la même propriété est décidable dans le cas où l'hypothèse FIFO est abandonnée.

Théorème 3.3.10 *Le problème de savoir si un HMSC régulier est implémentable par un CFM (non nécessairement FIFO) est indécidable.*

La preuve consiste à voir les CFMs non FIFO comme des réseaux de Petri et à se ramener à un problème d'accessibilité.

Dans [18], on généralise la notion d'implémentation en permettant de surcharger les messages avec une quantité (bornée) de données. Cette nouvelle notion d'implémentabilité étendue augmente strictement le pouvoir d'implémentabilité.

Théorème 3.3.11 *Tout HMSC localement coopératif a une implémentation étendue de taille exponentielle.*

Bibliographie

[1] Int. Telecom. Union recommendation Z.120, Message Sequence Charts, Geneva, 1999.

[2] R. Alur, K. Etessami et M. Yannakakis, Inference of Message Sequence Charts, in *22nd International Conference on Software Engineering*, pp. 304–313, 2000. Disponible sous `http://www.cis.upenn.edu/~alur/Icse00.pdf`.

[3] R. Alur, K. Etessami et M. Yannakakis, Realizability and verification of MSC graphs, in *ICALP '01*, vol. 2076, 2001. Disponible sous `http://www.cis.upenn.edu/~alur/Icalp01.pdf`.

[4] R. Alur et D. Peled, Undecidability of partial order logics, *Information Processing Letters* **69**,3 (1999). Disponible sous `http://www.cis.upenn.edu/~alur/Ipl99.ps.gz`.

[5] R. Alur et M. Yannakakis, Model Checking of Message Sequence Charts, in *Proc. 10th Intl. Conf. on Concurrency Theory*, pp. 114–129, Springer, 1999. Disponible sous `http://www.cis.upenn.edu/~alur/Concur99msc.ps.gz`.

[6] E. Andureau, P. Enjalbert et L. F. del Cerro, *Logique temporelle, Sémantique et validation de programmes parsallèles*, Masson, 1989.

[7] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci et P. Schnoebelen, *Systems and Software Verification Model-Checking Techniques and Tools*, Springer, 2001.

[8] J. Berstel, *Transductions and context-free languages*, *Leitfäden der Angewandten Mathematik und Mechanik [Guides to Applied Mathematics and Mechanics]* vol. 38, B. G. Teubner, Stuttgart, 1979.

[9] E. Clarke, E. Emerson et A. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems* **8**,2 (1986), 244–263.

[10] E. Clarke, O. Grumberg et D. Peled, *Model checking*, MIT Press, 1999.

[11] S. Demri et P. Schnoebelen, The Complexity of Propositional Linear Temporal Logics in Simple Cases, *Information and Computation* **174**,1 (2002), 84–103. Disponible sous `http://www.lsv.ens-cachan.fr/Publis/PAPERS/DS-ICOMP2001.ps`.

[12] V. Diekert et P. Gastin, Local Temporal Logic is Expressively Complete for Cograph Dependence Alphabets, in *LPAR'01*, Springer (éd.), pp. 55–69, *Lecture Notes in Artificial Intelligence* vol. 2250, 2001. Disponible sous `http://www.liafa.jussieu.fr/~gastin/Articles/Lpar01dg.html`.

[13] V. Diekert et Y. Métivier, *Partial Commutation and Traces*, vol. 3, ch. Partial Commutation and Traces, pp. 457–534, Springer, 1997. Disponible sous `ftp://ftp.informatik.uni-stuttgart.de/pub/techreports/theorie/handbook.ps`.

[14] V. Diekert et G. Rozenberg (éd.), *Book of Traces*, World Scientific, Singapore, 1995.

[15] J. Esparza et M. Nielsen, Decidability Issues for Petri Nets - a Survey, *Bulletin of the European Association for Theoretical Computer Science* **52** (1994), 245–262.

[16] P. Gastin, R. Meyer et A. Petit, A (non-elementary) modular decision procedure for LTrL., in *MFCS '98*, 1998. Disponible sous `http://www.lsv.ens-cachan.fr/Publis/PAPERS/GMP-mfcs98.ps`.

[17] P. Gastin et M. Mukund, An Elementary Expressively Complete Temporal Logic for Mazurkiewicz Traces, in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, P. Widmayer, F. Triguero, R. Morales, M. Hennessy, S. Eidenbenz et R. Conejo (éd.), pp. 938–949, 2002.

[18] B. Genest, A. Muscholl, H. Seidl et M. Zeitoun, Infinite-State High-Level MSCs: Model-Checking and Realizability, in *ICALP '02*, P. Widmayer et al. (éd.), *Lecture Notes in Computer Science* vol. 2380, Springer, 2002.

[19] J. Henriksen, M. Mukund, K. N. Kumar et P. Thiagarajan, On Message Sequence Graphs and Finitely Generated Regular MSC Languages, in *ICALP '00*, *Lecture Notes in Computer Science* n°1853, Springer, 2000. Disponible sous `http://www.brics.dk/~gulmann/Papers/icalp2000.ps.gz`.

[20] J. Henriksen, M. Mukund, K. N. Kumar et P. Thiagarajan, Regular Collections of Message Sequence Charts, in *MFCS'00*, *Lecture Notes in Computer Science* n°1893, Springer, 2000. Disponible sous `http://www.brics.dk/~gulmann/Papers/mfcs2000.ps.gz`.

[21] J. G. Henriksen, An Expressive Extension of TLC, in *ASIAN'99*, Springer (éd.), pp. 126–138, *Lecture Notes in Computer Science* vol. 1742, 1999. Disponible sous `http://www.brics.dk/~gulmann/Papers/asian99.ps.gz`.

[22] N. Klarlund, Progress measures for complementation of ω-automata with applications to temporal logic, in *Proc. of the 32nd Symposium on Foundations of Computer Science, Oct. 1-4, 1991, San Juan, Puerto Rico*, pp. 358–367, 1991.

[23] O. Kupferman et M. Y. Vardi, Weak alternating automata are not that weak, *TOCL* **2**,3 (2001), 408–429.

[24] M. Lohrey, Safe realizability of high-level message sequence charts, in *CONCUR'02*, pp. 177–192, *Lecture Notes in Computer Science* vol. 2421, 2002.

[25] Z. Manna et A. Pnueli, *The temporal logic of reactive and concurrent systems*, Springer Verlag, Berlin-Heidelberg-New York, 1991.

[26] A. Mazurkiewicz, Concurrent Program Schemes and their Interpretations, DAIMI Rep. PB n°78, Aarhus University, Aarhus, 1977.

[27] M. Michel, Complementation is more difficult with automata on infinite words. Manuscript., 1988.

[28] R. Morin, On Regular Message Sequence Chart Languages and Relationships to Mazurkiewicz Trace Theory, in *FoSSACS'01*, F. Honsell et M. Miculan (éd.), pp. 332–346, *Lecture Notes in Computer Science* vol. 2030, Springer, 2001.

[29] R. Morin, Recognizable sets of Message Sequence Charts, in *STACS'02*, H. Alt et A. Ferreira (éd.), pp. 523–534, *Lecture Notes in Computer Science* vol. 2285, Springer, 2002.

[30] A. Muscholl, Matching specifications for message sequence charts, in *Foundations of software science and computation structures (Amsterdam, 1999)*, pp. 273–287, *Lecture Notes in Comput. Sci.* vol. 1578, Springer, Berlin, 1999. Disponible sous `http://www.liafa.jussieu.fr/~anca/Publications/mus99fossacs.ps.gz`.

[31] A. Muscholl et D. Peled, Message sequence graphs and decision problems on Mazurkiewicz traces, Lecture Notes in Computer Science 1672, pp. 81-91, 1999., in *Proc. of MFCS'99*, 1999. Disponible sous `http://www.liafa.jussieu.fr/~anca/Publications/mp99mfcs.ps.gz`.

[32] A. Muscholl, D. Peled et Z. Su, Deciding properties for message sequence charts, in *Foundations of software science and computation structures (Lisbon, 1998)*, pp. 226–242, *Lecture Notes in Comput. Sci.* vol. 1378, Springer, Berlin, 1998. Disponible sous `http://www.liafa.jussieu.fr/~anca/Publications/mps98fossacs.ps.gz`.

[33] J. Queille et J. Sifakis, Specification and verification of concurrent systems in CESAR, in *Proc. Int. Symp on Programming*, pp. 337–351, *Lecture Notes in Computer Science* vol. 137, 1982.

[34] S. Safra, On the complexity of ω-automata, in *Proc. of the 29th Symposium on Foundations of Computer Science*, pp. 319–327, 1988.

[35] S. Safra, Exponential Determinization for omega-Automata with Strong-Fairness Acceptance Condition (Extended Abstract), in *ACM Symposium on Theory of Computing*, pp. 275–282, 1992.

[36] A. P. Sistla et E. M. Clarke, The complexity of propositional linear temporal logics, *Journal of the ACM* **32**,3 (1985), 84–103.

[37] W. Thomas, Automata on infinite objects, in *Handbook of Theoretical Computer Science*, J. van Leeuwen (éd.), vol. B, Formal models and semantics, pp. 135–191, Elsevier, 1990.

[38] W. Thomas, Languages, automata, and logic, in *Handbook of formal languages*, vol. 3, pp. 389–455, Springer, Berlin, 1997. Disponible sous `http://www-i7.informatik.rwth-aachen.de/~thomas/papers/survey.ps.gz`.

[39] W. Thomas, Complementation of Büchi Automata Revised, in *Jewels are Forever*, J. Karhumäki, H. A. Maurer, G. Paun et G. Rozenberg (éd.), pp. 109–120, Springer, 1999. Contributions on Theoretical Computer Science in Honor of Arto Salomaa.

[40] M. Y. Vardi, Alternating Automata and Program Verification, in *Computer Science Today*, pp. 471–485, 1995. Disponible sous `http://www.cs.rice.edu/~vardi/papers/vol1000.ps.gz`.

[41] I. Walukiewicz, Difficult configurations – On the complexity of LTrL, in *ICALP '98*, vol. 1443, 1998. Disponible sous `http://www.labri.fr/Perso/~igw/Papers/igw-trace-comp.ps`.

[42] W. Zielonka, Notes on finite asynchronous automata, **21** (1987), 99–135.