Spécification et Vérification des Systèmes Concurrents

(mémoire d'habilitation à diriger des recherches)

Ph. Schnoebelen [1]

[†] Laboratoire Spécification et Vérification
ENS de Cachan & CNRS UMR 8643
email : phs@lsv.ens-cachan.fr

[1]LSV, ENS de Cachan, 61 av. Pdt. Wilson, 94235 Cachan Cedex, FRANCE

Table des matières

Introduction générale

Ce mémoire d'habilitation à diriger des recherches propose un survol de mes travaux de recherche de ces dernières années.

J'ai choisi de mettre essentiellement l'accent sur mes contributions à la *théorie de la vérification* car il m'a semblé difficile de rassembler dans un même document des travaux issus de problématique plutôt éloignées telles que la sémantique du parallélisme, ou la mise en œuvre des langages de programmation. J'ai préféré ajouter une annexe à ce document (cf. page 89), où je présente très brièvement les travaux qui n'ont pas trouvé leur place dans le corps de ce mémoire.

La théorie de la vérification est un sujet de recherche très vivant, au moins depuis une dizaine d'années. Ceci s'explique en partie par les importantes applications pratiques de la vérification, applications qui fournissent un champ d'expérimentation pour la théorie et qui, en retour, posent sans cesse de nouvelles questions.

J'ai organisé cette présentation en trois parties :

1. La partie I est consacrée à la *logique temporelle*. Elle en rappelle les notions de base, qui me permettent ensuite de présenter plus en détail quelles extensions j'ai étudiées. Dans toute cette partie, on considère la logique temporelle comme le langage de spécification dans lequel on formalise les propriétés à vérifier, mais on s'abstient d'évoquer les techniques de vérification proprement dites. En conséquence, les questions traitées ont toutes à voir avec l'expressivité des divers formalismes manipulés.
2. La partie II traite du *model-checking*, c'est-à-dire la vérification automatique qu'un modèle donné satisfait une propriété (le plus souvent exprimée en logique temporelle). Cette partie ne considère que des modèles finis, et donc la question de la décidabilité ne se pose en général pas pour les logiques temporelles propositionnelles que nous manipulons. Le thème central est plutôt le coût, ou complexité intrinsèque, des problèmes de model-checking.
3. Enfin la partie III se consacre à la vérification des *systèmes infinis*. Confronté au grand nombre d'approches et de modèles différents auquel ce sujet se prête, j'ai choisi de mettre l'accent sur deux de mes contributions : la généralisation de la notion de systèmes bien structurés, et l'utilisation des automates d'arbres pour la vérification des processus PA.

Chacune de ces trois parties est construite de la même façon : une section introductive présente les notions et notation de bases utilisées dans la suite et donne les éléments bibliographiques que j'ai trouvés les plus pertinents. Les sections suivantes détaillent ensuite certaines de mes contributions pour des problèmes particuliers. En général, une section est essentiellement tirée d'une (ou plusieurs) de mes publications. Je termine chaque section par des notes bibliographiques utiles à qui voudrait en savoir plus sur le problème abordé par la section.

Ces trois parties ne présentent pas tous mes travaux sur la théorie de la vérification. L'annexe A les décrit brièvement, et plus généralement tous ceux parmi mes travaux qui n'ont pas trouvé leur place dans le corps de ce mémoire.

Première partie

Logique temporelle

Introduction

La logique temporelle est un des formalismes les plus utilisés pour la spécification et la vérification des systèmes réactifs. C'est un langage logique qui permet de *formaliser* certains aspects des systèmes, de *combiner* naturellement les propriétés, et de *raisonner* formellement sur les spécifications. Les algorithmes de model-checking (cf. Partie II) permettent ensuite de vérifier automatiquement qu'une modélisation du système satisfait bien les spécifications attendues.

Le terme *système réactif* désigne tout système dont l'objet est de maintenir une interaction avec son environnement (plutôt que de calculer un « résultat final ») [Pnu86]. Ce sont par exemple les systèmes d'exploitation, les protocoles, les jeux vidéo, les bases de données, les programmes distribués, etc. La spécification de leur comportement ne peut pas se résumer à une simple relation *hypothèses de début* $\Rightarrow$ *propriétés vérifiées à la fin* qu'on pourrait décrire en logique de Hoare.

Pour de tels systèmes, la logique temporelle apporte la possibilité de décrire simplement des relations complexes entre des événements situés à différents moments le long des exécutions, par exemple « *toute requête sera inévitablement suivie d'une réponse, sauf si une annulation a lieu entre temps* ». En pratique, la logique temporelle sert surtout à formaliser les *propriétés comportementales* du système (par exemple celles qu'on trouve dans un cahier des charges).

Nous n'avons pas l'ambition dans ce mémoire de justifier et d'expliquer plus avant l'intérêt pratique de la logique temporelle. Nous renvoyons pour cela au livre [SBB$^+$99].

Cette première partie voit essentiellement la logique temporelle comme un formalisme de spécification, et introduit la problématique de son expressivité. Le chapitre 1 rappelle les idées et définitions fondamentales, nécessaires pour exposer les résultats que nous voulons présenter plus loin. Nos notations sont standards et tout lecteur ayant travaillé avec la logique temporelle s'y retrouvera aisément.

Nous présentons ensuite deux extensions auxquelles nous avons contribué : $PCTL^*$, une logique temporelle *avec passé* (Chapitre 2), et $TCTL$ une logique temporelle *quantitative* (Chapitre 3).

Chapitre 1

Bases théoriques

Nous allons présenter CTL^*, une logique temporelle *propositionnelle* introduite par Emerson et Halpern [EH86] afin de réunir CTL et LTL en un même formalisme. (La plupart des résultats que nous mentionnerons dans la suite de ce mémoire porteront sur CTL ou LTL.)

1.1 La syntaxe de CTL^*

On considère un ensemble $Prop = \{A, B, \ldots\}$ de *propositions atomiques.*

Définition 1.1. (Syntaxe de CTL^*)
Les formules de CTL^* sont données par la grammaire :

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\psi \mid \mathsf{A}\varphi \mid A \mid \ldots$$

où $A, \ldots$ sont les propositions de $Prop$.

La formule $\mathsf{X}\varphi$ se lit « *next* φ », soit φ *est vraie dans l'état suivant.* $\varphi\mathsf{U}\psi$ se lit « φ *until* ψ », soit φ *jusqu'à ce que* ψ. $\mathsf{A}\varphi$ se lit « *all* φ », soit *dans toutes les exécutions,* φ.

On utilise librement les abréviations logiques habituelles $\top$, $\bot$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, et $\varphi \Leftrightarrow \psi$, ainsi que

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathsf{F}\varphi & \stackrel{\text{def}}{\equiv} & \top\mathsf{U}\varphi & \mathsf{G}\varphi & \stackrel{\text{def}}{\equiv} & \neg\mathsf{F}\neg\varphi \\
\stackrel{\infty}{\mathsf{F}}\varphi & \stackrel{\text{def}}{\equiv} & \mathsf{GF}\varphi & \stackrel{\infty}{\mathsf{G}}\varphi & \stackrel{\text{def}}{\equiv} & \mathsf{FG}\varphi \\
\varphi\mathsf{W}\psi & \stackrel{\text{def}}{\equiv} & \mathsf{G}\varphi \vee \varphi\mathsf{U}\psi & \mathsf{E}\varphi & \stackrel{\text{def}}{\equiv} & \neg\mathsf{A}\neg\varphi
\end{array}$$

$\mathsf{F}\varphi$ dit que φ est vraie dans un état futur (le long d'une exécution), le dual, $\mathsf{G}\varphi$, dit que φ est vraie dans tous les états futurs (d'une exécution). $\stackrel{\infty}{\mathsf{F}}\varphi$ dit que, le long d'une exécution, φ est vraie infiniment souvent, tandis que le dual, $\stackrel{\infty}{\mathsf{G}}\varphi$, dit que φ y est presque toujours vraie. La formule $\varphi\mathsf{W}\psi$ utilise W, le « *weak until* », et dit que φ est vraie tant que ψ n'est pas devenue vraie. W est un dual de U et la Définition 1.3 permettra de vérifier que

$$\varphi\mathsf{W}\psi \;\equiv\; \neg(\neg\psi\,\mathsf{U}\,\neg\varphi \wedge \neg\psi)$$

La *taille* d'une formule temporelle φ, notée $|\varphi|$, correspond à la longueur d'une écriture linéaire de φ. On pose donc

$$|A| \stackrel{\text{def}}{=} 1 \qquad \left.\begin{array}{r}|\varphi\wedge\psi|\\|\varphi\mathsf{U}\psi|\end{array}\right\} \stackrel{\text{def}}{=} 1 + |\varphi| + |\psi| \qquad \left.\begin{array}{r}|\neg\varphi|\\|\mathsf{X}\varphi|\\|\mathsf{A}\varphi|\end{array}\right\} \stackrel{\text{def}}{=} 1 + |\varphi| \tag{1.1}$$

La *hauteur temporelle* de φ, notée $h(\varphi)$ est une autre mesure de la complexité de φ. Il s'agit du nombre d'emboîtements de combinateurs temporels (X et U) dans φ, défini par

$$\begin{array}{rcl} h(A) & \stackrel{\text{def}}{=} & 0 \\ \left.\begin{array}{r} h(\neg\varphi) \\ h(\mathsf{A}\varphi) \end{array}\right\} & \stackrel{\text{def}}{=} & h(\varphi) \end{array} \qquad \begin{array}{rcl} h(\mathsf{X}\varphi) & \stackrel{\text{def}}{=} & 1 + h(\varphi) \\ h(\varphi \wedge \psi) & \stackrel{\text{def}}{=} & \max(h(\varphi), h(\psi)) \\ h(\varphi \mathsf{U} \psi) & \stackrel{\text{def}}{=} & 1 + \max(h(\varphi), h(\psi)) \end{array} \tag{1.2}$$

Par la suite, on étendra implicitement les définitions (1.1) et (1.2) à des logiques temporelles incluant d'autres combinateurs que ceux de CTL^*.

Dans certains cas, la formule temporelle φ est donnée sous forme d'un graphe acyclique, avec partage des sous-formules identiques. Cette deuxième façon de représenter φ conduit parfois à des différences importantes dans la taille des données et le coût des algorithmes. On écrit $|\varphi|_{\mathrm{g}}$ pour la taille « graphe » de φ, définie comme le nombre de sous-formules distinctes de φ. Pour tout φ on a l'encadrement

$$h(\varphi) \leq |\varphi|_{\mathrm{g}} \leq |\varphi| < 2^{|\varphi|_{\mathrm{g}}} \tag{1.3}$$

1.2 La sémantique de CTL^*

Les modèles de CTL^* et, plus généralement, des logiques temporelles et modales sont appelés *structures de Kripke*. Il s'agit en général d'un ensemble de mondes possibles reliés entre eux par certaines relations d'accessibilité ou de compatibilité [Kri63, HC96]. Mais dans le domaine de la vérification et du model-checking, une structure de Kripke n'est qu'un système de transitions (ou un automate), où les « mondes » sont les *états possibles* d'un système.

Définition 1.2. (Structure de Kripke)
Une structure de Kripke est un triplet $S = \langle Q, R, l\rangle$ où

1. $Q = \{q, q', \ldots\}$ *est un ensemble non-vide d'*états *;*
2. $R \subseteq Q \times Q$ *est une* relation de transition *;*
3. $l : Q \to 2^{Prop}$ *est un* étiquetage des états.

(Parfois on distingue un état initial dans S qui a alors la forme $\langle Q, R, l, q_0\rangle$.)

On exige que R soit *totale* [1], c.-à-d. $\forall q \exists q', (q, q') \in R$. On note $q \to q'$ quand $(q, q') \in R$ et on dit que q' est un successeur de q.

Une exécution de S est une suite infinie $\pi = (q_i)_{i\in\mathbb{N}}$, notée $q_0 q_1 \ldots$ d'états en succession : $q_i \to q_{i+1}$ pour tout $i \in \mathbb{N}$. On désigne par $\pi(i)$ le i-ème état de π (c.-à-d. q_i) et par π^i le i-ème suffixe (c.-à-d. $q_i q_{i+1} \ldots$). On écrit $\Pi(q)$ pour l'ensemble des exécutions partant de q, et $\Pi(S)$ pour l'ensemble de toutes les exécutions.

Les formules de CTL^* énoncent des propriétés d'une exécution π qui représente un état courant (ici $\pi(0)$) muni d'un futur possible. Les formules $\mathsf{A}\varphi$ ou $\mathsf{E}\varphi$ qui quantifient sur les futurs possibles énoncent des propriétés qui ne dépendent que de l'état courant.

[1] On peut justifier cette restriction en la rapprochant d'une technique classique de la théorie des automates où on se limite souvent à ne considérer que les seuls automates « complets ». Elle n'enlève rien à la puissance du modèle mais elle rend le cadre beaucoup plus léger et élégant : les exécutions sont toutes infinies. Par contre, quand il s'agit de modéliser des systèmes, cette restriction est parfois pénible et on peut regretter que certains outils (p.ex. SMV) ne considèrent pas les exécutions finies.

On écrit $\pi \models \varphi$ quand φ est vraie de π. La relation $\models$ est définie par induction sur la structure de φ :

Définition 1.3. (Sémantique de CTL^*)

1. $\pi \models A$ *ssi* $A \in l(\pi(0))$,
2. $\pi \models \neg\varphi$ *ssi* $\pi \not\models \varphi$,
3. $\pi \models \varphi \wedge \psi$ *ssi* $\pi \models \varphi$ *et* $\pi \models \psi$,
4. $\pi \models \mathsf{X}\varphi$ *ssi* $\pi^1 \models \varphi$,
5. $\pi \models \varphi \mathsf{U} \psi$ *ssi il existe* $i \geq 0$ *tel que* $\pi^i \models \psi$, *et* $\pi^j \models \varphi$ *pour* $0 \leq j < i$,
6. $\pi \models \mathsf{A}\varphi$ *ssi* $\pi' \models \varphi$ *pour tout* π' *tel que* $\pi'(0) = \pi(0)$.

La satisfaction d'une formule par un état est une notion dérivée : on écrit $q \models \varphi$ quand $\pi \models \varphi$ pour tout π de $\Pi(q)$, c.-à-d. quand toutes les exécutions issues de q vérifient φ.

Pour une structure S muni d'un état initial q_0, on dira que S *satisfait* φ, noté $S \models \varphi$, quand $q_0 \models \varphi$.

Remarque 1.4. Il y a là une différence importante, parfois source de confusion, entre notre point de vue (celui de la vérification) et le point de vue classique (celui des logiciens) où $S \models \varphi$ signifie qu'on a $q \models \varphi$ pour tout $q \in Q_S$.

Le point de vue de la vérification (appelé « *anchored* » en anglais [MP89]) peut se définir dans le cadre classique, p.ex. au moyen d'une proposition spéciale repérant l'état initial. Le point de vue classique ne peut pas s'exprimer aussi facilement dans notre cadre, où la satisfaction des propriétés ne peut pas être influencée par les états non atteignables à partir de l'état initial.

La différence entre les deux points de vue n'est pas seulement anecdotique : on peut la résumer lapidairement en disant que le point de vue classique considère la structure S comme l'objet étudié, alors que le point de vue de la vérification considère l'ensemble (arbre du comportement, ensemble de chemin, etc.) des comportements du système S. □

1.3 CTL, LTL et autres fragments de CTL^*

LTL est le fragment de CTL^* où l'on n'utilise pas les quantificateurs A et E. Pour une formule φ de LTL, savoir si $\pi \models \varphi$ ne dépend que de la suite des étiquetages $l(\pi(0))l(\pi(1)\ldots$ et pas des états qu'on pourrait atteindre en s'écartant de π. On dit que φ est une *formule de chemin.*

CTL est le fragment de CTL^* où l'on exige que tout combinateur U ou X soit immédiatement sous la portée d'un quantificateur A ou E. Ainsi $\mathsf{E}(A\mathsf{U}\mathsf{X}B)$ et $\mathsf{A}(\mathsf{F}A \wedge \mathsf{G}B)$ ne sont pas des formules CTL : dans $\mathsf{E}(A\mathsf{U}\mathsf{X}B)$ un U s'interpose entre E et X et dans $\mathsf{A}(\mathsf{F}A \wedge \mathsf{G}B)$ c'est un $\wedge$ qui s'interpose. Pour une formule φ de CTL, savoir si $\pi \models \varphi$ ne dépend que de l'état $\pi(0)$ et pas de l'exécution choisie. On dit que φ est une *formule d'état.* Pour les formules d'états, l'équivalence

$$q \models \neg\varphi \text{ ssi } q \not\models \varphi$$

est vérifiée (alors que ce n'est pas le cas pour une formule CTL^* en général).

Pour désigner commodément des fragments tels que LTL ou CTL, nous utiliserons les notations suivantes, faciles à mémoriser [2] [EH86, Eme90] :

[2] $L(\ldots)$ est une référence à « *linear-time* », $B(\ldots)$ à « *branching-time* ».

– $L(\mathsf{H}_1, \mathsf{H}_2, \ldots)$ désigne le fragment où les seuls combinateurs temporels autorisés sont H_1, H_2, ..., et où la quantification sur les chemins est interdite (les connecteurs booléens sont autorisés). Ainsi *LTL* est $L(\mathsf{X}, \mathsf{U})$ tandis que $L(\mathsf{U})$ (ou « $LTL - \mathsf{X}$ ») est la logique *sans next* recommandée dans [Lam83].
– $B(\mathsf{H}_1, \mathsf{H}_2, \ldots)$ désigne l'ensemble des formules où les seuls combinateurs temporels autorisés sont H_1, H_2, ..., et où chaque combinateur temporel doit être immédiatement sous la portée d'un quantificateur A ou E. Ainsi *CTL* est $B(\mathsf{X}, \mathsf{U})$ tandis que $B(\mathsf{X}, \mathsf{F})$ est la logique *UB* de [BPM83].

CTL et *LTL* sont les deux logiques temporelles les plus utilisées. Du point de vue de leur pouvoir d'expression, chacune a ses limites bien connues :
– *CTL* ne peut pas exprimer les propriétés d'équité ni certaines propriétés telles que $\mathsf{AF}(A \wedge \mathsf{X}A)$ (« *sur toutes les exécutions, on rencontrera inévitablement A dans deux états consécutifs* »). Ce sont surtout les propriétés d'équité qui font défaut en pratique [3].
– *LTL* est plus simple à manipuler (pas besoin de jongler avec les quantificateurs) et permet d'exprimer presque toutes les propriétés dont on a besoin en pratique, sauf l'atteignabilité potentielle (par exemple $\mathsf{AGEF}A$ en *CTL*).

Bien sûr, pour choisir un formalisme de spécification, d'autres critères que le pouvoir d'expression sont à prendre en compte. Nous comparons la complexité des problèmes de vérification pour *CTL* et *LTL* dans la partie II de ce mémoire.

1.4 Les questions d'expressivité en logique temporelle

Une question classique et très naturelle qui se pose au sujet de tout formalisme logique est celui de son *pouvoir d'expression*. La question générale qui nous intéresse alors est « *Quelles propriétés est-il possible d'exprimer dans tel ou tel formalisme ?* ». Cette question admet plusieurs sortes de réponses, chaque réponse apportant sa part d'information.

Prenons le cas d'une logique temporelle TL et voyons quelles sortes de résultats peuvent parfois être énoncés :

Formule inexprimable. D'un point de vue formel, on pourra montrer que telle propriété *ne peut pas s'exprimer* dans la logique TL. Par exemple, $\mathsf{E} \overset{\infty}{\mathsf{F}} A$ ne peut pas s'exprimer en *CTL* [EH86], ce qui permet de conclure que *CTL* ne permet pas d'exprimer les propriétés d'équité.

Comparaison de deux logiques temporelles. On notera TL1 $\leq$ TL2 quand toute formule φ_1 de TL1 est équivalente à une formule φ_2 de TL2, et on dira que TL2 est au moins aussi expressive que TL1, ou que TL1 n'est pas plus expressive que TL2. Par exemple, $CTL^+ \leq CTL$ [EH85] [4]. On en conclut que CTL^+ apporte des facilités d'écriture, mais pas de vrai pouvoir d'expression.

Comparaison avec un autre formalisme. On peut aussi comparer le pouvoir d'expression d'une logique temporelle avec un formalisme issu d'autres domaines (le plus souvent une logique de l'ordre, ou un formalisme d'automates).

Par exemple, un résultat célèbre dû à Kamp énonce que *LTL* exprime exactement les

[3] L'extension *FCTL*, c.-à-d. *CTL*+équité, est un excellent compromis [EL87]. Une autre solution largement répandue consiste à utiliser des structures de Kripke avec équité (comme fait SMV, par exemple).

[4] CTL^+ est l'extension de *CTL* où l'on autorise les combinaisons booléennes de formules de chemin dans la portée d'un quantificateur : $\mathsf{A}[(\mathsf{F}A \wedge \mathsf{G}B) \Rightarrow (C\mathsf{U}D)]$ est une formule de CTL^+.

mêmes propriétés que les formules à une variable libre de la logique du premier ordre $FO(<, P_A, \ldots)$ [Kam68].

Possibilité de décrire une structure de Kripke. Il est pertinent de comparer la logique temporelle aux équivalences comportementales utilisées en sémantique de la concurrence [5]. On se demande alors si « *être $\approx$-équivalent à S* » peut s'exprimer par une formule φ de TL. Si c'est le cas, on dit que *TL peut caractériser S modulo $\approx$*.

Expressibilité des combinateurs. Parfois on montrera qu'un *combinateur* (ou un contexte) donné est ou n'est pas exprimable. Par exemple, le théorème de Kamp tel qu'énoncé dans [Kam68] va au delà de l'équivalence « $LTL = FO(<, P_A, \ldots)$ » : il énonce que tout combinateur n-aire définissable en logique du premier ordre (en $FO(<, P_A, \ldots)$) est exprimable comme un contexte LTL.

Expressibilité succincte. Quand une formule (ou un combinateur) peut être exprimée dans TL, on peut parfois montrer qu'elle ne peut pas l'être *de manière succincte*. La possibilité théorique devient alors une impossibilité pratique. Par exemple, même si CTL^+ n'est pas plus expressif que CTL, une formule telle que $\mathsf{E}(\mathsf{F}A_1 \wedge \mathsf{F}A_2 \wedge \cdots \wedge \mathsf{F}A_n)$ n'a pas d'équivalent CTL de taille polynomiale [Wil99, AI01].

Pouvoir de distinction. Parfois on ne considère pas ce qu'expriment les formules mais plutôt la façon dont elles permettent de distinguer tel ou tel état, telle ou telle structure [6]. On dit que S et S' sont indistinguables par la logique TL, noté $S \equiv_{\mathrm{TL}} S'$, ssi $S \models \varphi \Leftrightarrow S' \models \varphi$ pour toute formule φ de TL. Alors l'équivalence $\equiv_{\mathrm{TL}}$ explicite quels détails des structures de Kripke sont visibles par TL.

Par exemple, l'équivalence $\equiv_{CTL}$ est la bisimilarité [7]. On en déduit que CTL n'est pas sensible au dépliage d'une structure de Kripke, et qu'elle ne permet donc pas d'exprimer qu'une structure est finie.

Une autre application de tels résultats est qu'ils caractérisent quelles modifications des structures de Kripke respectent la satisfaction des formules, ce qui a de nombreuses applications dans les techniques d'*abstraction correcte* visant à réduire la taille des systèmes examinés [CGL94, DGG97].

Expressivité pratique. Il reste à considérer le point de vue pratique ! Ici, on cherche à savoir si TL permet d'exprimer *simplement et naturellement* les propriétés *qu'on rencontre en pratique* [8] dans tel ou tel domaine d'application. Les critères sont alors plus subjectifs, mais il est souvent possible de dégager un consensus.

Par exemple, la logique LTL (même étendue au premier ordre) ne permet pas de spécifier le comportement d'entrée-sortie d'un buffer non-borné (c'est-à-dire ne permet pas d'écrire un équivalent formel de « *ce qui sort du buffer correspond à ce qui y est entré auparavant* ») [Koy89].

[5] Ces équivalences sont aussi utilisées dans des outils de vérification [Fer89, BRDV90, CPS93], ce qui augmente encore l'intérêt de la comparaison.

[6] Ce point de vue est très utilisé en sémantique de la concurrence, où les *caractérisations modales* des équivalences sémantiques leur apportent un éclairage supplémentaire. De fait, quand Hennessy et Milner ont proposé la bisimulation comme équivalence sémantique pour CCS, ils ont utilisé la logique modale *HML* pour justifier leur choix [HM85].

[7] Aucune des logiques temporelles mentionnées dans ce mémoire n'induit une équivalence plus fine que la bisimilarité.

[8] Plutôt que des propriétés bizarres inventées pour fournir des contre-exemples.

De même, les logiques pur-futur telles que LTL et CTL ne permettent pas de transcrire naturellement les propriétés exprimées en langue naturelle, comme on peut le faire avec des logiques temporelles combinant passé et futur [LS00a].

Dans le reste de cette première partie nous allons étudier, sous l'angle de l'expressivité, deux extensions de la logique temporelle. Ces deux extensions sont la *logique temporelle avec passé* (Chapitre 2) et une *logique temporelle quantitative* (Chapitre 3). Des motivations issues du model-checking font que nous considérerons essentiellement des extensions de CTL mais la plupart des résultats ont une portée plus générale comme nous nous attacherons à le signaler.

Dans le cadre des logiques temporelles avec passé, notre contribution principale est la recherche de théorèmes d'élimination du passé pour des logiques du temps arborescent (Cf. section 2.3).

Dans le cadre des logiques temporelles quantitatives, notre apport principal est l'étude de quelles contraintes temporisées permettent la formulation succincte de certaines propriétés (Cf. section 3.3).

1.5 Notes bibliographiques

C'est Pnueli qui a le premier suggéré d'utiliser la logique temporelle pour exprimer les propriétés comportementales des systèmes réactifs, et pour raisonner formellement à leur sujet [Pnu77, Pnu81]. Lamport a ensuite lancé le débat « temps linéaire vs. temps arborescent » [Lam80]. Dans ce débat, les points de vue ont surtout été guidés par des considérations et argumentations pratiques : « *comment exprimer l'équité en CTL ?* », ou « *quel est le coût du model-checking de LTL ?* », alors que les aspects plus fondamentaux « *quelle sémantique pour les processus concurrents ?* » ont été explorés par une autre communauté (cf. [Gla01]).

LTL date de [GPSS80], quand le U est venu compléter et remplacer le F [9]. CTL date de [CE81] et la logique de [QS83] lui est équivalente (mais [EC80] utilisait déjà une logique qui est essentiellement $ECTL$). CTL^* date de [EH83].

Les principaux résultats permettant de situer les pouvoirs d'expression respectifs de CTL et LTL (et CTL^*) se trouvent dans les travaux classiques [Lam80, BPM83, EH85, EH86].

C'est Pnueli qui a soulevé la question de ramener les problèmes d'équivalence avec une structure finie à des problèmes de logique temporelle [Pnu85]. Dans cette direction, le résultat le plus important est dû à Browne *et al.* [BCG88] qui montrent que la bisimilarité à une structure de Kripke finie est exprimable en CTL, et que $CTL-\mathsf{X}$ peut exprimer la bisimilarité modulo *stuttering* (ou « bisimilarité de branchement [GW96] »).

[9] Mais les théoriciens savaient depuis la thèse de Kamp [Kam68] que le combinateur U est nécessaire pour atteindre le pouvoir d'expression de la logique du premier ordre sur les ordres linéaires.

Chapitre 2

Logique temporelle avec passé

Les résultats présentés dans ce chapitre sont tirés de [LS95, LS00a].

2.1 Motivations

Les logiques telles que *CTL* et *LTL* sont des logiques *pur-futur* : les combinateurs X et U permettent de parler du futur de l'état courant. À l'inverse, il n'y a pas de combinateur qui permette de parler du passé de l'état courant.

Cette situation, typique en informatique, est un peu paradoxale. En effet, la logique temporelle trouve en partie ses origines dans l'étude et la formalisation des temps de la conjugaison, tels qu'on les utilise dans les langues naturelles, et où le passé est au moins aussi important que le futur. De fait, la logique temporelle vue par les logiciens traite souvent le passé et le futur à égalité [Pri67].

On peut avancer plusieurs raisons pour expliquer ce biais des informaticiens. D'abord il y a la pratique de la théorie des automates, un autre formalisme essentiellement pur-futur où l'on progresse le long d'un mot (ou d'un arbre) qui peut être infini à droite mais pas à gauche, alors que les formalismes issus de la logique de l'ordre traitent eux-aussi le passé et le futur à égalité. Ensuite, les algorithmes de model-checking simples et efficaces pour des logiques comme *CTL* ou *LTL* s'adaptent difficilement à des logiques mixant passé et futur. Finalement, on constate qu'il est très souvent possible d'écrire en logique pur-futur les propriétés qui semblent avoir besoin du passé (ceci dans la mesure où l'on spécifie des propriétés de comportement ayant un état initial bien défini).

2.1.1 Un exemple

Reprenons la propriété « *tout A sera finalement suivi d'un B* » (P1) et complétons là par un « *tout B a été précédé d'un A* » (P2).

En *CTL*, (P1) s'exprime directement par $\mathsf{AG}(A \Rightarrow \mathsf{AF}B)$. Exprimer (P2) en *CTL* est plus délicat. On y parvient par un raisonnement de contraposition : la négation de (P2) s'exprime « *il y a un B qui n'est pas précédé d'un A* », c.-à-d. « *il y a un B qui n'est précédé que de* $\neg A$ », soit $\mathsf{E}\neg A\mathsf{U}B$ (ou plus précisément $\mathsf{E}\neg A\mathsf{U}(B \wedge \neg A)$). Finalement, on exprime (P2) par $\neg\mathsf{E}\neg A\mathsf{U}B$.

Nous verrons plus loin (Théorème 2.6) que cet exemple est loin d'être un cas particulier : dans le cas de la logique *LTL*+Passé toute formule peut être exprimée par une formule *LTL* équivalente. Mais il nous faut auparavant donner quelques définitions.

2.2 La logique $PCTL^*$

$PCTL^*$ étend CTL^* par des combinateurs du passé.

Définition 2.1. (Syntaxe de $PCTL^*$)
Les formules de $PCTL^*$ sont données par la grammaire :

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\psi \mid \mathsf{A}\varphi \mid \underline{\mathsf{X}^{-1}\varphi} \mid \underline{\varphi\mathsf{S}\psi} \mid \underline{\mathsf{N}\varphi} \mid A \mid \ldots$$

où les combinateurs nouveaux ont été soulignés.

Informellement, X^{-1} (« *Previous* ») et S (« *Since* ») sont les versions « passé » de X et U. Plus précisément, $\mathsf{X}^{-1}\varphi$ se lit « φ *était vraie dans l'état immédiatement précédent* » et $\varphi\mathsf{S}\psi$ se lit « ψ *était vraie dans un état antérieur, et* φ *est vraie depuis lors* ».

F^{-1} est un combinateur dérivé de S via

$$\mathsf{F}^{-1}\varphi \stackrel{\text{def}}{=} \top\mathsf{S}\varphi$$

comme F est dérivé de U. Ainsi $\mathsf{F}^{-1}\varphi$ se lit « φ *était vraie dans un état antérieur* ». La propriété (P2) s'énonce maintenant directement par $\mathsf{AG}(B \Rightarrow \mathsf{F}^{-1}A)$.

2.2.1 Sémantique

Les formules de $PCTL^*$ s'interprètent dans un état courant, muni de son passé. Le passé est la suite des états qui ont conduit le système de son état initial à l'état courant. Notre définition du passé respecte trois principes, présentés et justifiés dans [LS00a] :

le passé est fini : c'est une hypothèse naturelle quand le passé correspond au calcul qui a conduit de l'état initial à l'état courant.

le passé est linéaire : c'est-à-dire déterminé. C'est une autre hypothèse naturelle. Le futur est non-déterministe et sera connu au fur et à mesure que le système avance dans son calcul. Quand le présent avance, le futur indéterminé devient un passé connu.

le passé est cumulatif : il augmente avec le temps. Les états rencontrés dans le passé restent des états passés, et on ajoute de nouveaux états plus récents au fur et à mesure de l'avancée du système.

Formellement, étant donnée une structure de Kripke $S = \langle Q, R, l\rangle$, une *histoire* est une suite finie $q_0q_1 \ldots q_n$ ($n \geq 0$) d'états de Q tels que $(q_i, q_{i+1}) \in R$ pour $0 \leq i < n$. On note $\sigma, \ldots$ pour les histoires, qu'il ne faut pas confondre avec les exécutions. Une histoire $\sigma = q_0q_1 \ldots q_n$ représente un état courant q_n avec son passé fixé : σ.

Les formules de $PCTL^*$ sont interprétées sur des exécutions avec un passé : soit $\pi = q_0q_1 \ldots$ une exécution dans S, soit $n \in \mathbb{N}$ un entier, le couple (π, n) représente l'état courant q_n à l'intérieur de π. Le passé est $q_0q_1 \ldots q_n$. À l'intérieur de π le futur est $q_nq_{n+1} \ldots$.

Définition 2.2. (Sémantique de $PCTL^*$)
On définit $\pi, n \models \varphi$ par induction sur la structure de φ :

1. $\pi, n \models A$ *ssi* $A \in l(\pi(n))$,
2. $\pi, n \models \neg\varphi$ *ssi* $\pi, n \not\models \varphi$,

3. $\pi, n \models \varphi \wedge \psi$ *ssi* $\pi, n \models \varphi$ *et* $\pi, n \models \psi$,
4. $\pi, n \models \mathsf{X}\varphi$ *ssi* $\pi, n+1 \models \varphi$,
5. $\pi, n \models \varphi \mathsf{U} \psi$ *ssi il existe* $i \geq n$ *tel que* $\pi, i \models \psi$, *et* $\pi, j \models \varphi$ *pour* $n \leq j < i$,
6. $\pi, n \models \mathsf{A}\varphi$ *ssi* $\pi', n \models \varphi$ *pour tout* π' *tel que* $\pi'(0) \ldots \pi'(n) = \pi(0) \ldots \pi(n)$,
7. $\pi, n \models \mathsf{X}^{-1}\varphi$ *ssi* $n > 0$ *et* $\pi, n-1 \models \varphi$,
8. $\pi, n \models \varphi \mathsf{S} \psi$ *ssi il existe* $i \leq n$ *tel que* $\pi, i \models \psi$, *et* $\pi, j \models \varphi$ *pour* $i < j \leq n$,
9. $\pi, n \models \mathsf{N}\varphi$ *ssi* $\pi^n, 0 \models \varphi$.

Remarquons que X^{-1} et S sont les pendants de X et U. Le passé est linéaire et fixé car le quantificateur A ne porte que sur le futur de π, n. Le passé est cumulatif car, dans le cas de combinateurs du futur X et U, les sous-formules sont évaluées sur des exécutions dont le passé prolonge $\pi(0) \ldots \pi(n)$.

2.2.2 Le combinateur N

N est utile dans les situations où un passé cumulatif ne convient pas. $\mathsf{N}\varphi$ se lit « *à partir de maintenant,* φ », ou « *en faisant table rase du passé,* φ ».

La signification de N est bien illustrée par un exemple. Imaginons que $\pi, n \models \mathsf{G}(A \Rightarrow \mathsf{F}^{-1}B)$. Alors tout A dans le futur doit être précédé d'un B quelque part dans son passé. Si maintenant $\pi, n \models \mathsf{NG}(A \Rightarrow \mathsf{F}^{-1}B)$, alors un B dans le passé d'un A doit se trouver après n, dans le futur de π, n. Si enfin $\pi, n \models \mathsf{GN}(A \Rightarrow \mathsf{F}^{-1}B)$, alors le B dans le passé d'un A ne peut même pas se trouver avant le A, il doit être dans son présent. En d'autres termes :

$$\begin{array}{rcl} \mathsf{G}(A \Rightarrow \mathsf{F}^{-1}B) & \equiv & \mathsf{F}^{-1}B \vee \mathsf{NG}(A \Rightarrow \mathsf{F}^{-1}B) \\ \mathsf{NG}(A \Rightarrow \mathsf{F}^{-1}B) & \equiv & \neg\big((\neg B)\mathsf{U}(A \wedge \neg B)\big) \\ \mathsf{GN}(A \Rightarrow \mathsf{F}^{-1}B) & \equiv & \mathsf{G}(A \Rightarrow B) \end{array}$$

Par ailleurs N vérifie un grand nombre d'équivalences, utiles pour simplifier l'écriture des formules :

Proposition 2.3. [LS95] *Pour toutes formules* φ, ψ *de* $PCTL^*$ *:*

$$\begin{array}{rclcrclcrcl} \mathsf{N}(\varphi \wedge \psi) & \equiv & \mathsf{N}\varphi \wedge \mathsf{N}\psi, && \mathsf{NN}\varphi & \equiv & \mathsf{N}\varphi, && \mathsf{NX}^{-1}\varphi & \equiv & \bot, \\ \mathsf{N}\neg\varphi & \equiv & \neg\mathsf{N}\varphi, && \mathsf{NA}\varphi & \equiv & \mathsf{AN}\varphi, && \mathsf{N}(\varphi\mathsf{S}\psi) & \equiv & \mathsf{N}\psi. \end{array}$$

(D'où l'on déduit l'important $\mathsf{NF}^{-1}\varphi \equiv \mathsf{N}\varphi$.)

2.2.3 Des fragments de $PCTL^*$

Les notations $B(\mathsf{H}_1, \ldots)$ et $L(\mathsf{H}_1, \ldots)$ se généralisent directement aux combinateurs X^{-1}, S et N. Comme on ne quantifie que sur le futur, nous écrirons les formules d'une logique comme $B(\mathsf{X}, \mathsf{U}, \mathsf{X}^{-1}, \mathsf{S})$ (c'est-à-dire « CTL+Since+Previous ») sans quantificateur au dessus des X^{-1} et des S. Par exemple, $\mathsf{AG}(A \Rightarrow \mathsf{F}^{-1}B)$ est une formule de $B(\mathsf{F}, \mathsf{F}^{-1})$.

La terminologie « formule pur-futur » s'applique à des formules ne contenant pas de combinateurs du passé. Symétriquement les formules pur-passé ne contiennent pas d'opérateurs du futur.

2.2.4 $PCTL^*$ pour la spécification

L'utilisation de $PCTL^*$ pour la spécification demande d'abord de définir des notions de satisfaction dérivées de la Définition 2.2 :

Définition 2.4. *Soit une formule φ de $PCTL^*$:*
Pour une exécution sans passé π, on dit que π satisfait φ, et on note $\pi \models \varphi$, ssi $\pi, 0 \models \varphi$.
On dit qu'un état q satisfait φ, noté $q \models \varphi$ ssi $\pi \models \varphi$ pour tout $\pi \in \Pi(q)$.
On dit qu'une structure de Kripke S avec un état initial q_{init} satisfait φ, noté $S \models \varphi$, ssi $q_{init} \models \varphi$.
On dit que φ est valide ssi $S \models \varphi$ pour toute structure S.

Ce sont ces notions qui sont pertinentes quand il s'agit de spécifier et de vérifier un système. En particulier, on énoncera que $S \models \varphi$ sans considérer aucun passé pour S : si $S \models \mathsf{AG}(A \Rightarrow \mathsf{F}^{-1}B)$, le passé dans lequel B est vraie est un passé de A, mais il se trouve dans le futur de S. On retrouve là une version avec passé de l'argumentation de la Remarque 1.4.

Un argument formel en faveur de ce point de vue est formalisé par le théorème suivant :

Théorème 2.5. [LS95] *Deux structures de Kripke S_1 et S_2 sont bisimilaires ssi pour toute formule φ de $PCTL^*$, $S_1 \models \varphi \Leftrightarrow S_2 \models \varphi$.*

2.2.5 Les critères d'équivalence

Classiquement on note $\varphi \equiv \psi$ quand φ et ψ sont équivalentes : dans notre cadre, l'équivalence en question est définie par :

$$\varphi \equiv \psi \overset{\text{def}}{\Leftrightarrow} \text{ pour tous } \pi \text{ et } n,\ \pi, n \models \varphi \text{ ssi } \pi, n \models \psi.$$

Cette équivalence a la très importante propriété d'être une congruence, ou d'être *substitutive* : $\varphi \equiv \varphi'$ implique $\psi[\varphi] \equiv \psi[\varphi']$ pour tout contexte $PCTL^*$ $\psi[.]$.

Toutefois cette équivalence ne correspond pas au critère de correction retenu dans la Déf. 2.4, c'est-à-dire qu'en général

$$\varphi \equiv \psi \underset{\not\Leftarrow}{\overset{\Rightarrow}{}} \models (\varphi \Leftrightarrow \psi)$$

Notons $\equiv_{\mathrm{i}}$ l'*équivalence initiale*, définie par $\varphi \equiv_{\mathrm{i}} \psi$ ssi $\models (\varphi \Leftrightarrow \psi)$. Cette équivalence est plus large que $\equiv$: par exemple on a

$$A \mathsf{U} B \underset{\not\equiv}{\overset{\equiv_{\mathrm{i}}}{}} \mathsf{F}\big(B \wedge \mathsf{G}^{-1}(A \vee B)\big).$$

La relation $\equiv_{\mathrm{i}}$ est le critère pertinent quand on veut comparer l'expressivité de deux logiques temporelles avec passé. C'est ce critère qui est sous-jacent au résultat classique selon lequel le passé n'apporte pas d'expressivité à la logique temporelle :

Théorème 2.6. [GPSS80, Gab89] *Toute formule de PLTL [1] est équivalente (au sens de $\equiv_{\mathrm{i}}$) à une formule de LTL.*

(De fait ce théorème devient trivialement faux si on l'énonce avec $\equiv$ au lieu de $\equiv_{\mathrm{i}}$.)

Le combinateur N permet de relier élégamment les équivalences $\equiv$ et $\equiv_{\mathrm{i}}$:

[1] C'est-à-dire $L(\mathsf{X}, \mathsf{U}, \mathsf{X}^{-1}, \mathsf{S})$, ou « LTL+Passé », une logique du temps linéaire.

Proposition 2.7. [LS95] *Pour toutes formules φ et ψ de PCTL*, on a*

$$\varphi \equiv_{\mathsf{i}} \psi \textit{ ssi } \mathsf{N}\varphi \equiv \mathsf{N}\psi.$$

2.3 L'élimination du passé

Le théorème de Gabbay est un résultat d'*élimination du passé*. Il montre qu'il est possible de remplacer toute formule φ avec passé (de $PLTL$) par une formule pur-futur φ^+ (de LTL) qui lui est équivalente (pour $\equiv_{\mathsf{i}}$). De plus il donne une méthode effective pour construire φ^+ à partir de φ.

La méthode utilisée par Gabbay passe par un théorème de séparation. On dit qu'une formule temporelle est (fortement) *séparée* si aucun combinateur du passé ne se trouve dans la portée d'un combinateur du futur, et vice versa. Ainsi A et $\mathsf{F}A \vee \mathsf{X}^{-1}B$ sont séparées, mais $\mathsf{F}(A \vee \mathsf{X}^{-1}B)$ et $\mathsf{X}^{-1}\mathsf{F}(A \vee B)$ ne le sont pas. Dans le cas de $PLTL$, une formule séparée est une combinaison booléenne de formules pur-futur et de formules pur-passé. Gabbay montre alors

Théorème 2.8. [GPSS80, Gab89] (Séparation forte pour $PLTL$) *On peut transformer toute formule φ de PLTL en une formule séparée φ^{sep} équivalente.*

De $\varphi \equiv \varphi^{sep}$ on déduit $\varphi \equiv_{\mathsf{i}} \mathsf{N}\varphi^{sep}$ et il est immédiat de simplifier $\mathsf{N}\varphi^{sep}$ en une formule pur-futur φ' au moyen des équivalences données dans la Proposition 2.3. Ainsi le Théorème 2.6 est un corollaire direct du Théorème 2.8.

2.3.1 Séparation pour $PCTL^*$

Dans le cas de logiques du temps arborescent, cette approche ne peut pas réussir avec une telle notion de formule séparée. Par exemple, $\mathsf{F}^{-1}\mathsf{EX}A$ dit qu'un état passé a un successeur vérifiant A. Aucune formule séparée ne peut énoncer la même chose. En effet, une formule séparée peut parler de tous les états futurs possibles, ainsi que de tous les états passés, mais elle ne peut pas parler des autres états (en particulier, pas des successeurs des états passés).

C'est pourquoi nous avons été conduits à introduire une notion de séparation plus faible :

Définition 2.9. [LS95] *Une formule temporelle φ est* faiblement séparée *si dans φ aucun combinateur du passé ne se trouve dans la portée d'un combinateur du futur (NB : sans exiger le vice versa).*

Ainsi les formules fortement séparées sont faiblement séparées *a fortiori* et $\mathsf{X}^{-1}\mathsf{F}(A \vee B)$ est un exemple de formule faiblement séparée.

On peut alors démontrer

Théorème 2.10. [LS95] (Séparation faible pour $PCTL^*$) *On peut transformer toute formule φ de PCTL* en une formule faiblement séparée φ^{sep} équivalente.*

Notons que la séparation faible suffit à déduire, via la Proposition 2.3, le corollaire recherché :

Théorème 2.11. [LS95] *Toute formule φ de PCTL* est équivalente (au sens de $\equiv_{\mathsf{i}}$) à une formule φ' de CTL* qui peut être construite effectivement.*

2.3.2 Séparation pour $PCTL$

Dans le cas de $PCTL$ il n'est pas possible de séparer les formules en restant dans $PCTL$. Plus précisément il n'est pas possible de traduire $CTL + \mathsf{X}^{-1}$ (c.-à-d. $B(\mathsf{X}, \mathsf{U}, \mathsf{X}^{-1})$) dans CTL

modulo $\equiv_{\rm i}$ [LS95]. Par exemple la formule $\mathsf{EG}(A \vee \mathsf{X}^{-1}A)$ est équivalente initialement à $\mathsf{EG}(A \vee \mathsf{X}A)$, une formule inexprimable en CTL [EH86].

De même $CTL + \mathsf{S}$ (c.-à-d. $B(\mathsf{X}, \mathsf{U}, \mathsf{S})$) ne peut pas être traduite dans CTL [LS95].

Remarquablement, ce constat ne ferme pas la porte à toute possibilité de traduction vers CTL. En effet, nous avons été surpris de pouvoir montrer le résultat suivant :

Théorème 2.12. [LS95] (Séparation faible pour $PCTL + \mathsf{F}^{-1} + \mathsf{N}$) *On peut transformer toute formule φ de $PCTL + \mathsf{F}^{-1} + \mathsf{N}$ en une formule faiblement séparée φ^{sep} équivalente.*

Ce qui conduit au corollaire :

Théorème 2.13. [LS95] *Toute formule de $PCTL + \mathsf{F}^{-1} + \mathsf{N}$ est équivalente (au sens de $\equiv_{\rm i}$) à une formule de $PCTL$.*

Ce résultat est robuste [Lar94] : il peut s'étendre à CTL+Équité, et tolère le « F^{-1} strict », noté F_{+}^{-1} et défini par

$$\mathsf{F}_{+}^{-1}\varphi \stackrel{\text{def}}{=} \mathsf{X}^{-1}\mathsf{F}^{-1}\varphi.$$

Dans [LS00a] nous montrons comment aller au-delà de $PCTL + \mathsf{F}^{-1} + \mathsf{N}$ et comment il est possible de traduire dans CTL un fragment de $PCTL^*$ où l'on autorise certaines occurrences de X^{-1} et S dans les parties droites de EU.

2.4 Notes bibliographiques

La logique temporelle telle qu'introduite par Prior [Pri67] traite passé et futur symétriquement. C'est dans [GPSS80] qu'apparaît l'argument selon lequel on peut se dispenser du passé.

Toutefois, l'intérêt *pratique* des combinateurs du passé est reconnu. Et d'un point de vue théorique, ils permettent dans [LPZ85, MP90, CMP92] de classifier élégamment les formules temporelles en une hiérarchie allant des propriétés de sûreté aux propriétés de réactivité.

Pour les logiques du temps arborescent, le choix de la sémantique des combinateurs du passé est moins évident. Nous avons justifié nos choix dans [LS00a]. La logique $PCTL^*$ a été introduite dans [LS95]. Elle généralise à la fois le CTL_{lp} de [KP95] (où ne figure pas le N) et le $PCTL^*$ de [HT87] (où les quantificateurs A et E oublient le passé et sont équivalents à notre NA et NE).

Pour la logique du temps linéaire, le théorème de séparation passé-futur date de [Kam68]. [GPSS80] en donne une preuve plus simple, qui sera détaillée dans [Gab89]. [MMKR94] donne une variante du résultat de Gabbay. Pour les logiques du temps arborescent, le premier théorème de séparation passé-futur date de [LS95].

Chapitre 3

Logique temporelle quantitative

Les résultats présentés dans ce chapitre sont tirés de [LST02].

3.1 Motivations

Les logiques telles que *CTL* et *LTL* sont des logiques temporelles qui n'ont qu'une vision réduite du temps. Essentiellement, elles peuvent exprimer des contraintes sur l'ordonnancement temporel d'événements (la distinction *avant/après*). Elles ne permettent pas (ou pas facilement) d'exprimer une information quantitative sur les durées. Or il existe de nombreuses situations où une spécification temporelle n'a guère d'intérêt sans contrainte quantitative. Par exemple, dire que « *le salaire sera versé un jour* » sans garantir un délai maximum retire beaucoup à la spécification.

Ce constat est ancien. Dans le cadre de la logique temporelle, il a conduit au développement de ce qu'on appelle les logiques *temporisées*, qui permettent d'énoncer des contraintes quantitatives sur la durée qui sépare deux événements. En général ces logiques temporisées sont interprétées sur des modèles eux-mêmes temporisés, par exemple des systèmes de transitions temporisés décrits par des automates temporisés.

3.2 La logique $TCTL_{\mathrm{s}}$

Une façon naturelle d'introduire des contraintes sur les durées est d'adjoindre aux combinateurs temporels une *contrainte de durée*. Par exemple, on écrira $\mathsf{F}_{<5}A$ pour dire que A aura lieu dans le futur ($\mathsf{F}A$) *avant 5 unités de temps*. Dans le même esprit, on utilise $\mathsf{F}_{>k}$, $\mathsf{F}_{=k}$ où $k \in \mathbb{N}$, et plus généralement $\mathsf{U}_{\sim k}$ où $\sim \in \{<, >, =\}$. Pour $\varphi \mathsf{U}_{\sim k} \psi$, la contrainte « $\sim k$ » porte sur les moments où ψ est vérifiée. Comme d'habitude, φ doit être vérifiée en attendant.

Dans le cadre des automates temporisés, l'utilisation de telles contraintes conduit à la logique *TCTL* de Alur, Courcoubetis et Dill (utilisée dans l'outil Kronos [Yov97]). L'utilisation d'automates temporisés fait à la fois la force et l'inconvénient de cette approche. C'est un inconvénient car ils induisent un saut de complexité dans le model-checking (qui devient PSPACE-complet [ACD93, AL99]) et, surtout, ils imposent au modélisateur [1] de changer de classe des modèles, de passer des structures de Kripke bien connues à des automates plus

[1] P. ex. l'ingénieur soucieux d'utiliser une méthode formelle rigoureuse lors de la phase de conception des systèmes.

complexes, munis d'invariants, de gardes et d'horloges, et dont la sémantique n'est pas encore complètement stable [Lab98].

3.2.1 Une logique quantitative basée sur les structures de Kripke

Il est possible d'énoncer des propriétés quantitatives tout en restant dans le cadre des structures de Kripke classiques :

Définition 3.1. (Syntaxe de $TCTL_\mathrm{s}$)
Les formules de $TCTL_\mathrm{s}$ sont données par la grammaire :

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{EX}\varphi \mid \mathsf{E}\varphi\mathsf{U}_I\psi \mid \mathsf{A}\varphi\mathsf{U}_I\psi \mid A \mid \dots$$

où I est n'importe quelle partie finie ou cofinie de $\mathbb{N}$.

La contrainte I est dénotée par une union finie $[a_1, b_1[\cup[a_2, b_2[\cup \cdots \cup [a_n, b_n[$ d'intervalles de $\mathbb{N}$ avec $0 \leq a_1 < b_1 < a_2 < b_2 < \cdots < a_n < b_n \leq \omega$.

En pratique, on note « $< k$ » pour la contrainte $[0, k[$, « $> k$ » pour $[k+1, \omega[$ et « $= k$ » pour $[k, k+1[$. On retrouve les combinateurs CTL dans le cas $I = [0, \omega[$.

La formule $\varphi\mathsf{U}_I\psi$ se lit « *il existe un futur dans I où ψ est vérifiée, et φ est vérifiée jusque là* ». $\mathsf{F}_I\varphi$ est une abréviation pour $\top\mathsf{U}_i\varphi$ (« *il existe un futur dans I où φ est vérifiée* »). $\mathsf{G}_I\varphi$ est une abréviation pour $\neg\mathsf{F}_I\neg\varphi$, soit « *$\varphi$ est vérifiée dans tous les futurs appartenant à I* ».

3.2.2 Sémantique

Les formules de $TCTL_\mathrm{s}$ s'interprètent sur des structures de Kripke classiques. Toutefois, il existe une proposition, notée *tick*, dont la signification est spéciale et décrit la façon dont le temps avance. Intuitivement, un état porte l'étiquette *tick* ssi c'est un état dans lequel on observe que le temps vient d'avancer d'une unité [2]. Cette approche, dite à « horloge fictive », correspond à une modélisation basée sur une discrétisation du système étudié.

Formellement, on adapte la Définition 1.3 par

Définition 3.2. (Sémantique de $TCTL_\mathrm{s}$)
On définit $q \models \varphi$ par induction sur la structure de φ :

1. $q \models A$ *ssi* $A \in l(q)$,
2. $q \models \neg\varphi$ *ssi* $q \not\models \varphi$,
3. $q \models \varphi \wedge \psi$ *ssi* $q \models \varphi$ *et* $q \models \psi$,
4. $q \models \mathsf{EX}\varphi$ *ssi il existe* $(q, q') \in R$ *tel que* $q' \models \varphi$,
5. $q \models \mathsf{E}\varphi\mathsf{U}_I\psi$ *ssi il existe* $\pi \in \Pi(q)$ *tel que* $\pi \models \varphi\mathsf{U}_I\psi$,
6. $q \models \mathsf{A}\varphi\mathsf{U}_I\psi$ *ssi* $\pi \models \varphi\mathsf{U}_I\psi$ *pour tout* π *de* $\Pi(q)$,
7. $\pi \models \varphi\mathsf{U}_I\psi$ *ssi il existe* $i \geq 0$ *tel que* $Time(\pi, i) \in I$, $\pi^i \models \psi$, *et* $\pi^j \models \varphi$ *pour* $0 \leq j < i$.

où $Time(\pi, i) \stackrel{def}{=} |\{0 < j \leq i \mid \pi(j) \models tick\}|$ *est le nombre de ticks rencontrés en allant de* $\pi(0)$ *à* $\pi(i)$.

[2] En pratique, il est plus naturel de mettre les propositions *tick* sur les transitions que sur les états. L'approche que nous avons retenue est essentiellement équivalente, avec l'avantage de simplifier et d'alléger le formalisme.

La clause 7 est celle qui prend en compte les contraintes. On notera qu'elle ne demande pas que la première occurrence de ψ respecte I. Ainsi, $\mathsf{AF}_{\geq 5}A$ signifie qu'inévitablement on rencontrera A, et qu'on le rencontrera *entre autres* à des dates supérieures à 5. Il n'est pas exigé qu'on ne rencontre A qu'à la date 5 au plus tôt.

3.3 Le pouvoir d'expression des contraintes

Évaluer le pouvoir d'expression de $TCTL_{\mathrm{s}}$ demande d'abord de choisir quels sont les critères pertinents. Les questions les plus intéressantes concernent le pouvoir d'expression de divers types de contraintes (dans un cadre CTL), et pas celui de tel ou tel combinateur dans un cadre temporisé.

Dans la suite de cette section, nous allons montrer comment les contraintes simples « $< k$ », « $> k$ » et « $= k$ » apportent toutes un réel supplément d'expressivité, tandis que les contraintes à base d'unions d'intervalles se ramènent aux contraintes simples. Ces résultats ne sont pas vraiment surprenants, mais ils demandent parfois des énoncés savamment ciselés et des preuves délicates à construire.

Notre premier résultat rappelle que $TCTL_{\mathrm{s}}$ et CTL permettent d'écrire les mêmes propriétés, de sorte que toute mesure de l'apport de $TCTL_{\mathrm{s}}$ devra se faire en termes d'« expression succincte » :

Théorème 3.3. [LST02] *Pour toute formule φ de $TCTL_{\mathrm{s}}$, il existe une formule φ' de CTL telle que $\varphi \equiv \varphi'$.*

L'idée sous-jacente à cette construction est facile à comprendre sur un exemple. Considérons la formule $\mathsf{EF}_{=2}A$: elle exprime qu'il existe un chemin le long duquel A est rencontré après exactement 2 *tick*'s. On peut énoncer cette propriété au moyen de deux *until*s emboîtés :

$$\mathsf{EXE}\neg tick\,\mathsf{U}\left(\begin{array}{l}\overbrace{tick}^{\text{premier } tick}\wedge\\ \mathsf{EXE}\neg tick\,\mathsf{U}\left(\begin{array}{l}\overbrace{tick}^{\text{second } tick}\wedge\\ A\vee\mathsf{EXE}\neg tick\mathsf{U}(A\wedge\neg tick)\end{array}\right)\end{array}\right) \tag{3.1}$$

Il resterait alors à donner une construction générale, mais il est plus simple de voir le Théorème 3.3 comme un corollaire du Théorème 3.10.

La traduction suggérée par notre exemple n'est pas succincte : on exprime un $\mathsf{EF}_{=k}$ par k *until*s emboîtés. De fait, on peut montrer que toute traduction a un coût au moins exponentiel, même quand on ne considère que des contraintes $< k$:

Lemme 3.4. [LST02] *Soit φ une formule de CTL telle que $\varphi \equiv \mathsf{EF}_{<k}A$. Alors $h_{\mathsf{X}}(\varphi) \geq k$ (où $h_{\mathsf{X}}(\varphi)$ dénote la hauteur maximale de X emboîtés dans φ).*

Ainsi non seulement $|\varphi|$ et $|\varphi|_{\mathrm{g}}$ sont en $\Omega(k)$, c.-à-d. en $\Omega(2^{|\mathsf{EF}_{<k}A|})$, mais le nombre maximal de X emboîtés est minoré par k. Un corollaire de ce résultat est que l'utilisation de X est cruciale pour traduire $TCTL_{\mathrm{s}}$ dans CTL. Ce qui conduit au

Théorème 3.5. [LST02] *$TCTL_{\mathrm{s}} - \mathsf{X}$ est plus expressif que $CTL - \mathsf{X}$.*

Notons $TCTL_{\mathrm{s}}^{<}$ (resp. $TCTL_{\mathrm{s}}^{<,>}$; resp. $TCTL_{\mathrm{s}}^{<,>,=}$) le fragment de $TCTL_{\mathrm{s}}$ où ne sont autorisées que les contraintes de la forme $< k$ (resp. $< k$ et $> k$; resp. $< k$, $> k$ et $= k$).

Le Théorème 3.3 affirme que $TCTL_s^{<,>,=}$ peut se traduire dans CTL. Il semble toutefois que $TCTL_s^{<,>,=}$ peut être exponentiellement plus succinct que $TCTL_s^{<,>}$, qui peut lui-même être exponentiellement plus succinct que $TCTL_s^{<}$, qui peut lui-même être exponentiellement plus succinct que CTL [3]. Ainsi la prise en considération de contraintes de plus en plus variées augmente de façon bien réelle le pouvoir d'expression de $TCTL_s$.

En revanche

Théorème 3.6. [LST02] *Pour toute formule φ de $TCTL_s$ il existe dans $TCTL_s^{<,>,=}$ une formule φ' équivalente à φ et telle que $|\varphi'|_g$ est en $O(|\varphi|_g)$.*
De plus, φ' peut être construite en temps linéaire.

3.4 La logique $TCTL_q$

$TCTL$ est une façon naturelle d'étendre CTL en permettant de préciser des contraintes quantitatives sur les modalités *until*, mais il existe une autre façon, plus générale, d'exprimer le même genre de contraintes. Cette autre approche est basée sur la notion d'*horloges de formule* (ou « *freeze quantifiers* ») introduite dans [AH94].

Nous allons introduire ici $TCTL_q$, une version de $TCTL$ à base d'horloges de formule, et qui reprend l'esprit de $TCTL_s$: on utilise des structures de Kripke classique dans lesquelles on compte les *tick*s.

3.4.1 Une logique quantitative avec horloges de formule

On considère un ensemble $Cl = \{x, y, \dots\}$ de variables appelées *horloges de formule*.

Définition 3.7. (Syntaxe de $TCTL_q$)
Les formules de $TCTL_q$ sont données par la grammaire :

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{EX}\varphi \mid \mathsf{E}\varphi\mathsf{U}\psi \mid \mathsf{A}\varphi\mathsf{U}\psi \mid \underline{x \mathtt{\ in\ } \varphi} \mid \underline{x < k} \mid A \mid \dots$$

où les constructions nouvelles ont été soulignées.

Intuitivement, une formule telle que $x \mathtt{\,in\,} \varphi$ signifie que φ est vérifiée en supposant que dans l'état courant on démarre un compteur (un chronomètre) x. Les propositions $x < k$ présentes dans φ seront évaluées à vrai ou faux suivant la valeur qu'aura le chronomètre x dans les états rencontrés. (Notons qu'il est possible de combiner librement les litéraux $x < k$, de sorte que $TCTL_q$ peut exprimer $x > k$ et $x = k$.)

Par exemple, $q \models x \mathtt{\ in\ } \mathsf{AG}(A \Rightarrow x < 4)$ signifie que dans tout état q' atteint à partir de q (via un chemin π), si $q' \models A$ alors $x < 4$. La valeur courante de x sera le nombre de *tick* rencontrés entre q et q' le long de π. Finalement, $x \mathtt{\ in\ } \mathsf{AG}(A \Rightarrow x < 4)$ aurait pu s'écrire en $TCTL_s$ par $\mathsf{AG}_{\geq 4}\neg A$.

3.4.2 Sémantique

L'utilisation d'horloges de formule conduit à la notion classique d'occurrences libres ou liées des variables, la liaison étant réalisée par la construction « $x \mathtt{\ in\ } \_$ ».

[3]Parmi ces observations, seules les deux dernières sont prouvées dans [LST02], alors que la séparation de $TCTL_s^{<,>,=}$ et $TCTL_s^{<,>}$ en termes d'expression succincte reste une conjecture non démontrée.

Les formules *fermées* sont les formules sans variable libre ; ce sont elles qui expriment les propriétés des structures de Kripke. Mais la sémantique des formules est définie en supposant une valuation $v : Cl \to \mathbb{N}$ pour les horloges.

Formellement, on adapte la Définition 1.3 en introduisant une notion de $q, v \models \varphi$.

Définition 3.8. (Sémantique de $TCTL_{\mathrm{q}}$)
On définit $q, v \models \varphi$ par induction sur la structure de φ :

1. $q, v \models A$ *ssi* $A \in l(q)$,
2. $q, v \models \neg\varphi$ *ssi* $q, v \not\models \varphi$,
3. $q, v \models \varphi \wedge \psi$ *ssi* $q, v \models \varphi$ *et* $q, v \models \psi$,
4. $q, v \models \mathsf{EX}\varphi$ *ssi il existe* $(q, q') \in R$ *tel que* $q', v' \models \varphi$ *où* $v' = v + 1$ *si* $q' \models tick$, $v' = v$ *sinon.*
5. $q, v \models \mathsf{E}\varphi\mathsf{U}\psi$ *ssi il existe* $\pi \in \Pi(q)$ *tel que* $\pi, v \models \varphi\mathsf{U}\psi$,
6. $q, v \models \mathsf{A}\varphi\mathsf{U}\psi$ *ssi* $\pi, v \models \varphi\mathsf{U}\psi$ *pour tout* π *de* $\Pi(q)$,
7. $\pi, v \models \varphi\mathsf{U}\psi$ *ssi il existe* $i \geq 0$ *tel que* $\pi^i, v + Time(\pi, i) \models \psi$, *et* $\pi^j, v + Time(\pi, j) \models \varphi$ *pour* $0 \leq j < i$,
8. $q, v \models x \,\mathtt{in}\, \varphi$ *ssi* $q, v[x/0] \models \varphi$.
9. $q, v \models x < k$ *ssi* $v(x) < k$.

où l'on utilise la notation classique pour les variants

$$(v+m)(x) \stackrel{def}{=} v(x) + m \quad v[x/0](y) \stackrel{def}{=} \begin{cases} 0 & \text{si } x \neq y, \\ v(y) & \text{sinon.} \end{cases}$$

La clause 7 est celle qui décrit comment les horloges sont mises à jour pour rendre compte de l'écoulement du temps. Les valeurs des horloges sont utilisées dans la clause 9 et la clause 8 introduit une nouvelle horloge, mise à zéro dans l'état courant.

Évidemment, si x n'est pas libre dans φ, alors la valeur de $v(x)$ ne joue aucun rôle quant à la véracité de $q, v \models \varphi$. Comme il est classique en logique, nous profiterons parfois de ce fait pour n'écrire que la partie pertinente de v, voire pour omettre v quand φ est fermée.

3.5 Le pouvoir d'expression des horloges de formule

Notons tout d'abord que $TCTL_{\mathrm{q}}$ permet facilement d'exprimer les propriétés écrites en $TCTL_{\mathrm{s}}$: par exemple, $\mathsf{A}A\mathsf{U}_{[a,b[}B$ s'écrira $x \,\mathtt{in}\, \mathsf{A}A\mathsf{U}(B \wedge a \leq x < b)$. D'où le

Théorème 3.9. [LST02] *À toute formule φ de $TCTL_{\mathrm{s}}$ on peut associer une formule équivalente φ_c de $TCTL_{\mathrm{q}}$. De plus φ_c n'utilise qu'une horloge et $|\varphi_c|$ est en $O(|\varphi|)$.*

Notons que ce résultat n'a besoin de presque aucune hypothèse sur la façon dont nous mesurons le temps. Il reste valide dans tout modèle à base de système de transitions temporisé : [Alu91] l'énonce pour des automates temporisés.

$TCTL_{\mathrm{q}}$ permet aussi d'écrire des propriétés quantitatives riches combinant les délais s'étant écoulés entre plusieurs états particuliers. Considérons par exemple la propriété exprimée par :

$$x \,\mathtt{in}\, \mathsf{EF}(A \wedge \mathsf{EF}(B \wedge x = 10)). \qquad \text{(P)}$$

(P) exprime qu'on peut atteindre un (état vérifiant) A à partir duquel on peut atteindre un B, lequel se situe 10 unités de temps après l'état courant. Il y a donc dans cet exemple une contrainte « $x = 10$ » *portant sur la suite de deux* EF *emboîtés*. Par ailleurs les contraintes peuvent être combinées librement, mêler plusieurs horloges, et s'appliquer à des suites de combinateurs temporels avec alternation arbitraire de E et A.

Il est donc surprenant d'avoir :

Théorème 3.10. [LST02] *Pour toute formule φ de $TCTL_{\mathrm{q}}$, il existe une formule φ' de CTL telle que $\varphi \equiv \varphi'$.*

La preuve repose sur deux particularités de $TCTL_{\mathrm{q}}$. D'abord on n'autorise que certaines formes particulières de contraintes, de sorte que les valeurs contenues dans v ne sont pertinentes que jusqu'à un certain point. Pour $m \in \mathbb{N}$ et deux valuations v et v', on définit $v \equiv_m v'$ par

$$v \equiv_m v' \overset{\text{def}}{\Leftrightarrow} \forall x \in Cl, \begin{cases} v(x) = v'(x), \text{ ou bien} \\ v(x) \geq m \text{ et } v'(x) \geq m. \end{cases}$$

On dit alors que v et v' sont *égales jusqu'à m*. Remarquons que $v \equiv_m v'$ implique $v[x/0] \equiv_m v'[x/0]$ et $v + m' \equiv_{m+m'} v' + m'$ (et donc $v + m' \equiv_m v' + m'$).

Définissons alors $M(\varphi)$ comme la plus grande constante k apparaissant dans les contraintes $x_i < k_i$ de φ. Par exemple $M(\varphi) = 10$ dans l'exemple (P).

Le fait que les valeurs de v ne sont pertinentes que jusqu'à un certain point s'énonce alors :

Lemme 3.11. [LST02] *Si $v \equiv_m v'$ et $m \geq M(\varphi)$ alors pour tout état q de toute structure, $q, v \models \varphi$ ssi $q, v' \models \varphi$.*

On peut alors montrer le lemme suivant, qui a le Théo.3.10 comme corollaire immédiat :

Lemme 3.12. [LST02] *Pour toute formule φ de $TCTL_{\mathrm{q}}$, pour toute valuation de v, il existe une formule φ^v de CTL qui lui est équivalente au sens où $q, v \models \varphi$ ssi $q \models \varphi^v$ pour tout état q de toute structure.*

$$\begin{aligned} (\varphi \wedge \psi)^v &\overset{\text{def}}{=} \varphi^v \wedge \psi^v & \qquad (x < k)^v &\overset{\text{def}}{=} \begin{cases} \top \text{ si } v(x) < k, \\ \bot \text{ sinon} \end{cases} \\ (\neg\varphi)^v &\overset{\text{def}}{=} \neg(\varphi^v) & & \\ (A)^v &\overset{\text{def}}{=} A & (x \,\mathtt{in}\, \varphi)^v &\overset{\text{def}}{=} \varphi^{v[x/0]} \end{aligned}$$

$$(\mathsf{AF}\varphi)^v \overset{\text{def}}{=} \begin{cases} \mathsf{AF}\varphi^v \text{ si } v+1 \equiv_{M(\varphi)} v, \\ \varphi^v \vee \mathsf{AX}\Big[\mathsf{A}(\neg tick)\mathsf{U}\Big[(\neg tick \wedge \varphi^v) \vee (tick \wedge (\mathsf{AF}\varphi)^{v+1})\Big]\Big] \\ \text{sinon} \end{cases}$$

$$(\mathsf{E}\varphi \,\mathsf{U}\, \psi)^v \overset{\text{def}}{=} \begin{cases} \mathsf{E}\varphi^v \,\mathsf{U}\, \psi^v \text{ si } v+1 \equiv_{M(\varphi,\psi)} v, \\ \psi^v \vee \left[\varphi^v \wedge \mathsf{EX}\begin{pmatrix} \mathsf{E}\,(\varphi^v \wedge \neg tick) \\ \mathsf{U}\,(\psi^v \wedge \neg tick) \vee \big(tick \wedge (\mathsf{E}\varphi \,\mathsf{U}\, \psi)^{v+1}\big) \end{pmatrix}\right] \\ \text{sinon} \end{cases}$$

FIG. 3.1 – Définition inductive de φ^v.

Démonstration. La construction de φ^v se fait par induction sur la structure de φ et sur v (voir Figure 3.1). On se convainc que la définition récursive est bien fondée en vérifiant qu'à chaque fois que φ^v est définie en termes de $\varphi^{v'}$ alors $v' = v + 1$ et $v \not\equiv_{M(\varphi)} v'$, ce qui ne peut pas se poursuivre indéfiniment.

La propriété « $q, v \models \varphi$ ssi $q \models \varphi^v$ » se vérifie alors par induction sur la définition de φ^v. $\square$

Les règles pour AF et $\mathsf{E\_U}$ comptent les *ticks* à la manière de (3.1). Là aussi le X est indispensable et $TCTL_{\mathrm{q}} - \mathsf{X}$ ne peut pas être traduit en $CTL - \mathsf{X}$ (NB : mais peut être traduit en $TCTL_{\mathrm{s}} - \mathsf{X}$).

Ainsi, toute formule $TCTL_{\mathrm{q}}$ peut s'exprimer en CTL et, *a fortiori* en $TCTL_{\mathrm{s}}$. Mais reprenons notre exemple $x\ \mathtt{in}\ \mathsf{EF}(A \wedge \mathsf{EF}(B \wedge x = 10))$: écrire cette propriété en $TCTL_{\mathrm{s}}$ est assez délicat. Puisque le A a lieu avant 10 unités de temps, on voudrait écrire $\mathsf{EF}_{\leq 10}(A \wedge \mathsf{EF}_{??}B)$ mais la contrainte du second EF devrait prendre en compte la date effective du A, qui n'est pas connue. Finalement on ne voit guère d'autre solution que $\bigvee_{i=0}^{10} \mathsf{EF}_{=i}(A \wedge \mathsf{EF}_{=10-i}B)$.

Cet exemple suggère que $TCTL_{\mathrm{q}}$ apporte un réel pouvoir d'expression, mais il est difficile de démontrer formellement cette intuition. C'est un travail en cours, mené avec F. Laroussinie et M. Turuani. D'ores et déjà, on peut dire que s'il existait un algorithme en temps polynômial pour la traduction de $TCTL_{\mathrm{q}}$ en $TCTL_{\mathrm{s}}$, alors les Théorèmes 7.3 et 7.5 (Chapitre 7) impliqueraient P=PSPACE. Mais notre conjecture concerne la taille des formules après traduction, pas le coût algorithmique leur construction.

3.6 Notes bibliographiques

Il existe de nombreux travaux sur la prise en compte du « temps réel » dans les modèles informatiques, allant du modèle de Merlin pour les réseaux de Petri aux algèbres de processus temporisés (voir [BM01] et les références qu'il cite).

Dans le cadre de la logique temporelle pour la spécification et la vérification, la plupart des logiques temps-réel s'inspirent des travaux de Alur et Henzinger [AH92, AH94]. Un résultat fondamental est l'algorithme de model-checking de [ACD93] pour la logique $TCTL$ interprétée sur des automates temporisés (à la Alur et Dill [AD94]). Cet algorithme est basé, comme d'autres algorithmes d'analyse des automates temporisés, sur la notion de *région* définie par des contraintes d'horloge et est à l'origine d'outils tels que Kronos [Yov97] ou Uppaal [LPY97].

La logique $RTCTL$ introduite dans [EMSS92] est très semblable à notre $TCTL_{\mathrm{s}}^{<}$. On trouve des extensions de $RTCTL$ dans la littérature. P.ex., la logique de [ET97] permet de compter le nombre d'actions avec des expressions régulières et [ET99] permet la coexistence de différentes échelles de temps. [YMW97] propose une extension de UB par des horloges de formules, où il est permis des contraintes riches sur les valeurs des horloges. Aucun de ces travaux ne cherche à mesurer formellement le pouvoir d'expression des différents formalismes temporels introduits.

Deuxième partie

Model-checking

Introduction

Le *model-checking* est la vérification automatique qu'un modèle opérationnel S satisfait ou non une propriété φ (exprimée dans un langage formel).

En principe, le model-checking permet de vérifier a posteriori qu'un système satisfait bien les propriétés comportementales attendues.

L'intérêt pratique de cette technique vient de l'existence d'outils de model-checking (des *model-checkers*) performants, qui peuvent grandement aider à découvrir les erreurs présentes dans des systèmes (ou algorithmes, ou schémas, ...) réactifs ou concurrents. Quand les erreurs ont été identifiées et corrigées, le model-checking peut garantir que certaines propriétés sont satisfaites *dans tous les cas*, ce qu'une approche basée sur le test ne peut en principe pas garantir.

Un intérêt du model-checking est son côté « presse-bouton ». En principe, l'ingénieur qui veut prouver la correction d'un système n'a qu'à énoncer formellement, le plus souvent dans un langage déclaratif, les spécifications à satisfaire. Ensuite le model-checker fonctionne automatiquement, contrairement aux approches s'appuyant sur des outils de preuve assistée, qui demandent beaucoup de guidage de la part de l'utilisateur.

Ces dernières années, l'utilisation du model-checking s'est beaucoup développée, surtout dans les phases de conception de systèmes critiques (pour lesquels les exigences de correction justifient d'apprendre à utiliser une méthode nouvelle). On peut s'attendre à ce qu'à moyen terme le model-checking permette systématiquement de concevoir plus vite et à moindre coût des systèmes plus corrects, comme c'est déjà le cas dans le domaine de la conception de circuits intégrés. En effet, les conditions nécessaires à ce développement semblent réalisées : les techniques algorithmiques progressent toujours, les connaissances nécessaires à leur bonne utilisation se répandent, et on commence à trouver des model-checkers vendus (et maintenus) par des compagnies privées.

Nous renvoyons au livre [SBB$^+$99] pour plus de détails sur l'utilisation pratique du model-checking.

Dans cette deuxième partie nous allons essentiellement considérer le problème du model-checking sous l'angle de la complexité. Le chapitre 4 met cette problématique en place et rappelle les principaux résultats classiques du domaine : idées des algorithmes et mesure de complexité structurelle des problèmes. Notre ambition est de poser rapidement les bases qui nous permettront de présenter des résultats plus nouveaux dans les sections suivantes. Le chapitre 5 présente nos résultats sur le model-checking de LTL ; le chapitre 6 traite du model-checking de $PCTL^*$; enfin le chapitre 7 est consacré au model-checking de $TCTL_{\mathrm{q}}$ et $TCTL_{\mathrm{s}}$.

Chapitre 4

Algorithmes et complexité

La définition du model-checking que nous avons donnée dans l'introduction de cette seconde partie est très générale et recouvre un grand nombre de situations qui sortent du cadre standard. Il s'agit par exemple des cas où S est un modèle de taille infinie, ou bien où la spécification formelle φ n'est pas une formule de logique temporelle [1]. Dans toute la suite de cette deuxième partie nous ne considérons que *le model-checking de formules de logiques temporelles sur des structures de Kripke finies*.

D'un point de vue théorique, un algorithme de model-checking reçoit comme données d'entrée une structure de Kripke $S = \langle Q, R, l, q_0 \rangle$ et une formule de logique temporelle φ. Il répond ensuite « *vrai* » si $S \models \varphi$, « *faux* » sinon.

Pour mesurer le coût d'un algorithme de model-checking, il nous faut une définition raisonnable de la taille de ses arguments. Nous renvoyons à la section 1.1 pour la taille des formules, et nous définissons la taille $|S|$ d'une structure de Kripke en supposant que l'ensemble *Prop* est fixé, et en posant $|S| \stackrel{\text{def}}{=} |Q| + |R|$ où $|Q|$ est le nombre d'états de S et $|R|$ le nombre de paires (q, q') de R.

4.1 Les algorithmes de model-checking

4.1.1 Model-checking de *CTL*

L'algorithme classique pour le model-checking de *CTL* termine en temps bilinéaire $O(|S| \times |\varphi|)$ [CES86, SBB$^+$99].

Son fonctionnement repose sur des procédures de marquage [2] du graphe sous-jacent à S, une procédure pour chaque combinateur de *CTL*. Par exemple, si φ est de la forme $\mathsf{A}\varphi_1\mathsf{U}\varphi_2$, on va marquer chaque état q de S par un booléen $q.\varphi$ disant si $q \models \varphi$. Pour cela, on effectue le marquage de S pour φ_1 et φ_2. Ensuite les $q.\varphi$ sont calculés par une procédure `Mark_A_U` qui utilise les $q.\varphi_1$ et $q.\varphi_2$ et n'a besoin que d'un temps $O(|S|)$.

[1] Par exemple, les énoncés de la forme « *S est équivalent au système S_{spec}* » rentrent dans ce cadre du model-checking élargi.

[2] [CG87] contient une description claire de cette idée. [SBB$^+$99] propose une description alternative, inspirée par [AC88], plus simple que [CG87, CES86] dans la mesure où elle ne repose pas sur le calcul des composantes fortement connexes d'un sous-graphe de S.

4.1.2 Model-checking de *LTL*

L'existence d'un algorithme quadratique (ou bilinéaire) pour le model-checking de *CTL*, algorithme simple à décrire qui plus est, a longtemps plaidé en faveur de *CTL* dans le débat entre les partisans de *CTL* et ceux de *LTL*.

Pour *LTL*, la construction d'un model-checker présentait des difficultés spécifiques. On savait le problème décidable (depuis [Pnu77]) et même PSPACE-complet [SC85], jusqu'à ce que [LP85] présente un algorithme de coût $O(|S| \times 2^{|\varphi|})$ et avance l'idée que c'est la taille de S, pas celle de φ, qui est le vrai goulot d'étranglement du model-checking. Ainsi le coût du model-checking de *LTL* serait comparable à celui de *CTL* : linéaire dans la taille de S.

À la même époque, Vardi et Wolper proposaient un algorithme de même complexité à base d'automates de Büchi [WVS83, VW86a, VW86b]. Aujourd'hui les algorithmes les plus faciles à décrire sont basés sur des automates de mots infinis (par exemple, des automates de Büchi) : pour vérifier si $S \models \varphi$, on construit un automate de Büchi $B_{\neg\varphi}$ acceptant les chemins satisfaisant $\neg\varphi$ (les « mauvaises exécutions ») et on examine le produit synchronisé $S \times B_{\neg\varphi}$. Si le système obtenu n'a aucun chemin infini, alors toutes les exécutions de S sont correctes et on déduit que $S \models \varphi$. On peut construire le produit $S \times B_{\neg\varphi}$ (et y chercher un chemin infini) en temps $O(|S| \times |B_{\neg\varphi}|)$. Le coût de l'algorithme vient de la taille de $B_{\neg\varphi}$. La méthode de construction de $B_{\neg\varphi}$ est simple à comprendre, surtout si on utilise des automates alternants comme étape intermédiaire [Var96], mais $B_{\neg\varphi}$ peut avoir un nombre d'états en $2^{O(|\varphi|)}$).

En pratique, une façon simple d'implémenter ces algorithmes est celle de [CGH97] qui ramène directement le problème du model-checking symbolique de *LTL* en un problème de model-checking symbolique pour *CTL*+équité, pour lequel on peut utiliser par exemple SMV.

4.1.3 Model-checking de *CTL**

Au minimum, le model-checking de *CTL** doit rencontrer les difficultés du model-checking de *CTL* et de *LTL*. Il se trouve qu'il est facile de décrire un algorithme étendant simplement l'algorithme pour *LTL*, et demandant lui aussi un temps $O(|S| \times 2^{|\varphi|})$ [Eme90, Théorème. 6.26].

On peut donner l'idée de cet algorithme sur un exemple : supposons qu'on veuille vérifier $S \models \varphi$ pour $\varphi = \mathsf{G}(\mathsf{X}B \Rightarrow \mathsf{AF}(A \wedge \mathsf{X}A))$. On commence par isoler $\varphi_0 = \mathsf{F}(A \wedge \mathsf{X}A)$ et on écrit φ comme $\varphi_1[\mathsf{A}\varphi_0]$ avec $\varphi_1[\_] = \mathsf{G}(\mathsf{X}B \Rightarrow \_)$. L'algorithme de model-checking pour *LTL* permet, en examinant $S \times B_{\neg\varphi_0}$, de trouver tous les états de S qui vérifient la formule (d'état) $\mathsf{A}\varphi_0$. On introduit une nouvelle proposition A_0 et on construit S' en étendant S par l'étiquetage : $A_0 \in l(q) \overset{\text{def}}{\Leftrightarrow} q \models \mathsf{A}\varphi_0$. (En général, on introduira une proposition par occurrence d'un quantificateur A (ou E) dans la formule considérée.) Alors $S \models \varphi$ ssi $S' \models \varphi_1[A_0]$, et $\varphi_1[A_0]$ est une formule *LTL*.

Cet algorithme est, comme on le voit, presque aussi simple que le model-checking de *LTL* [3] et on peut s'étonner qu'il ne soit (à notre connaissance) implémenté dans aucun outil. L'explication tient peut-être à la difficulté qu'il y a à expliquer *CTL** à des utilisateurs potentiels, ou au fait que *CTL** ne se prête pas facilement à certains raffinements du model-checking *LTL* (par exemple le model-checking à la volée, ou la production de chemins expliquant pourquoi une formule n'est pas satisfaite dans un modèle donné).

[3] [CGH97] fait la même remarque.

4.2 La complexité du model-checking

Comme dans d'autres domaines, la mesure de la complexité théorique des problèmes sert de guide à la recherche des algorithmes les plus efficaces. Elle permet aussi, face à un problème intrinsèquement difficile, d'orienter la recherche vers des variantes (ou restrictions) plus simples du problème.

4.2.1 Model-checking de *LTL*

Concernant la complexité du model-checking, le résultat le plus connu concerne *LTL* :

Théorème 4.1. [SC85] *Le model-checking de LTL est PSPACE-complet.*

En effet, d'une part on peut implémenter l'algorithme vu plus haut pour *LTL* sans construire explicitement $B_{\neg\varphi}$. À la place, on engendre des transitions de $B_{\neg\varphi}$ au fur et à mesure qu'on en a besoin pour explorer les boucles dans $S \times B_{\neg\varphi}$. Tout ceci peut se faire en espace polynômial.

D'autre part on peut montrer que le problème est PSPACE-difficile, même quand on ne considère que des fragments réduits de *LTL* tels que $L(\mathsf{X}, \mathsf{F})$ ou $L(\mathsf{U})$ ou même d'autres exemples comme nous verrons au chapitre 5.

4.2.2 Model-checking de *CTL*

Le résultat suivant est beaucoup moins connu, probablement parce qu'il n'y a guère lieu de « justifier » le coût de l'algorithme de model-checking pour *CTL*.

Théorème 4.2. *Le model-checking de CTL est P-complet.*

Il s'agit d'un de ces *folk theorems* pour lesquels on ne peut clairement identifier aucun premier auteur. En fait le model-checking est déjà P-complet pour les fragments $B(\mathsf{X})$ et $B(\mathsf{F})$ de *CTL*.

Démonstration. Une preuve directe passe par une réduction du problème CIRCUIT-VALUE en un problème de model-checking.

Nous illustrons la construction sur un exemple. Soit C le circuit booléen de la figure 4.1 :

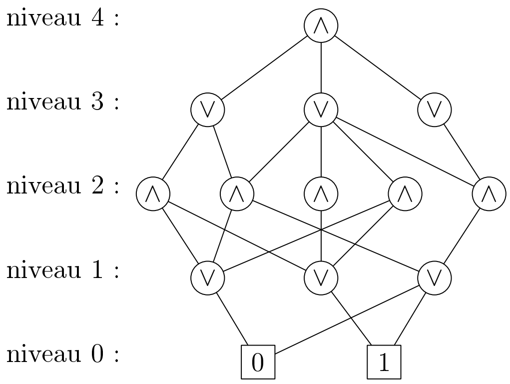

FIG. 4.1 – C, une instance de CIRCUIT-VALUE.

c'est un circuit *synchronisé* (les noeuds d'un niveau n ne dépendent que de noeuds du niveau $n-1$), *alternant* (on passe d'un noeud conjonctif à un noeud disjonctif et *vice versa*), et *monotone* (pas de négation). L'évaluation de tels circuits est exactement le problème SAMCVP, notoirement P-complet [GHR95, p. 123].

Si on voit maintenant C comme une structure de Kripke S_C dont l'état initial serait le noeud au sommet de C, on a

$$C \text{ s'évalue à 1 ssi } S_C \models \mathsf{AX\ EX\ AX\ EX}\ 1$$

ce qui fournit la réduction recherchée. Notons que dans cette construction la formule temporelle dépend de C (de sa profondeur plus précisément) sans quoi le résultat serait en contradiction avec le Théorème 4.4.(*2*). □

4.2.3 Model-checking de CTL^*

Comme on s'y attend après la section 4.1, le Théorème 4.1 conduit à :

Théorème 4.3. [EH86] *Le model-checking de CTL^* est PSPACE-complet.*

Il suffit ici de vérifier que l'algorithme présenté dans la section 4.1 peut s'adapter de façon à fonctionner en espace polynômial, ce qui est immédiat : si φ contient n quantificateurs A ou E, le model-checking de φ se réduit à $1 + n \times |S|$ questions de model-checking pour des formules de LTL.

4.2.4 Les deux paramètres de la complexité du model-checking

À première vue, la comparaison des Théorèmes 4.1 et 4.2 semble plaider en faveur du choix de CTL comme logique temporelle préférée pour le model-checking. (De fait, ce discours est répandu.)

En réalité, il convient d'examiner de plus près le coût de l'algorithme de model-checking pour LTL. Dans le coût $O(|S| \times 2^{|\varphi|})$ donné dans la section 4.1.2, les deux paramètres S et φ n'ont pas le même poids, et le model-checking de spécifications LTL est tout à fait envisageable pour des formules de taille limitée et des structures de Kripke de taille très importante. Justement, il se trouve que l'utilisation pratique des model-checkers mène souvent à vérifier des formules de taille réduite sur des structures de grande taille, permettant à Pnueli et Lichtenstein de défendre l'idée que LTL n'est pas plus coûteux que CTL dans les applications de model-checking [LP85].

Du point de vue de la théorie de la complexité, ces idées conduisent à la définition des complexités *en programme* et *en formule*, c'est-à-dire la complexité quand la structure de Kripke S est le seul paramètre qui varie (φ est considérée fixée), ou quand la formule φ est le seul paramètre. Ces notions, introduites par [VW86a] en adaptant au model-checking les idées de [Var82] pour l'interrogation des bases de données, permettent de mesurer l'impact de chacun des deux paramètres sur la complexité, en considérant pour chaque paramètre quels sont les algorithmes les plus efficaces (éventuellement au détriment de la façon dont est manipulé l'autre paramètre).

On a alors les résultats suivants :

Théorème 4.4. (Cf. Fig. 4.2)

1. *Pour LTL, le model-checking en programme est NLOGSPACE-complet, le model-checking en formule est PSPACE-complet.*
2. *Pour CTL, le model-checking en programme est NLOGSPACE-complet, le model-checking en formule est en temps $O(|\varphi|)$ ou en espace $O(\log|\varphi|)$.*

3. *Pour CTL^*, le model-checking en programme est NLOGSPACE-complet, le model-checking en formule est PSPACE-complet.*

	Complexité	en programme	en formule
LTL	PSPACE-complet	NLOGSPACE-complet	PSPACE-complet
CTL	P-complet	NLOGSPACE-complet	LOGSPACE
CTL^*	PSPACE-complet	NLOGSPACE-complet	PSPACE-complet

FIG. 4.2 – Complexité du model-checking : les deux paramètres.

Remarque 4.5. Dire que la complexité en programme (resp. en formule) est X-complet veut dire que :

1. pour toute formule φ (resp. programme S) le problème de savoir si $S \models \varphi$, un problème qui n'a que S (resp. φ) comme argument, est dans X ;
2. et *il existe* des formules (resp. des programmes) pour lesquelles le problème est X-dur. □

On peut expliquer sommairement comment s'obtiennent ces résultats.

Complexité en programme, borne inf. : Dans chacune des trois logiques on peut écrire une formule fixée énonçant l'atteignabilité d'un état (ou sa négation pour LTL), un problème notoirement NLOGSPACE-complet [Jon75].

Complexité en programme, borne sup. : Pour une formule LTL fixée φ, savoir si un état d'une structure de Kripke S donnée vérifie $\mathsf{E}\varphi$ n'est qu'un problème d'atteignabilité dans le produit synchronisé de S et de l'automate de Büchi reconnaissant les modèles de φ, ce qu'on peut résoudre en NLOGSPACE [VW94, Théo. 2.4].

Pour CTL^* (et CTL) le même raisonnement s'applique [KVW00, Théo. 5.8] mais il faut cette fois se ramener à la décision du vide des automates alternants hésitants à une lettre.

Complexité en formule, cas de CTL : L'algorithme en temps $O(|\varphi|)$ est celui du cas non-paramétré, l'algorithme en espace $O(\log|\varphi|)$ est donné plus bas (Théo. 4.6).

Complexité en formule, cas de LTL et CTL^* : Il existe une structure S pour laquelle le model-checking d'une formule φ de LTL se ramène à la validité de φ et est donc PSPACE-dur : c'est la structure qui contient toutes les valuations possibles et tous les chemins entre ces valuations. Cet argument suppose que l'ensemble des propositions apparaissant dans φ est fixé (et fini) lui aussi, mais le problème de la validité est PSPACE-complet même pour des formules n'utilisant que deux propositions [DS02].

À notre connaissance, le résultat suivant, qui minimise l'impact de $|\varphi|$ (au détriment de celui de $|S|$) n'a jamais été indiqué dans la littérature. Notons que l'algorithme de [KVW00, Théo. 5.7] utilise un espace en $O\big(|\varphi| \times \log^2(|\varphi| \times |S|)\big)$.

Théorème 4.6. *Le model-checking CTL peut être résolu en espace $O(|S| \times \log|\varphi|)$.*

La preuve utilise des techniques très classiques. Soit $S = \langle Q, R, l\rangle$. Posons $n = |Q|$. Pour chaque $q \in Q$, pour chaque sous-formule ψ de φ, et pour chaque $0 \leq k \leq n$ on note $v_q[k, \psi]$ la variable booléenne disant si q satisfait le $k$ième dépliage de φ. Formellement on définit

$$\begin{aligned}
v_q[k, A] &:= \begin{cases} \top & \text{si } A \in l(q), \\ \bot & \text{sinon} \end{cases} && (4.1) \\
v_q[k, \mathsf{EX}\psi] &:= \bigvee_{(q,q') \in R} v_{q'}[n, \psi] && (4.2) \\
v_q[k, \psi_1 \wedge \psi_2] &:= v_q[n, \psi_1] \ \wedge \ v_q[n, \psi_2] && (4.3) \\
v_q[k, \neg\psi] &:= \neg v_q[n, \psi] && (4.4) \\
v_q[0, \mathsf{E}\psi_1\mathsf{U}\psi_2] &:= v_q[n, \psi_2] && (4.5) \\
v_q[0, \mathsf{A}\psi_1\mathsf{U}\psi_2] &:= v_q[n, \psi_2] && (4.6) \\
v_q[k+1, \mathsf{E}\psi_1\mathsf{U}\psi_2] &:= v_q[n, \psi_2] \vee \Big(v_q[n, \psi_1] \wedge \bigvee_{(q,q') \in R} v_{q'}[k, \mathsf{E}\psi_1\mathsf{U}\psi_2] \Big) && (4.7) \\
v_q[k+1, \mathsf{A}\psi_1\mathsf{U}\psi_2] &:= v_q[n, \psi_2] \vee \Big(v_q[n, \psi_1] \wedge \bigwedge_{(q,q') \in R} v_{q'}[k, \mathsf{A}\psi_1\mathsf{U}\psi_2] \Big) && (4.8)
\end{aligned}$$

Il s'agit d'une définition bien fondée puisque chaque $v_q[k, \psi]$ ne dépend que de $v_{q'}[k', \psi']$ pour lesquels soit ψ' est une sous-formule stricte de ψ, soit $\psi' = \psi$ et $k' < k$.

Lemme 4.7. *Pour tous q et ψ on a :*

$$v_q[n, \psi] = \top \text{ ssi } q \models \psi. \tag{4.9}$$

Démonstration. La preuve est essentiellement la preuve de correction des algorithmes classiques de model-checking *CTL*. Elle se fait par induction structurelle sur ψ et nous nous contentons ici de rappeler comment traiter le cas où ψ est de la forme $\mathsf{A}\psi_1\mathsf{U}\psi_2$, en supposant (hypothèse d'induction no. 1) que le lemme a déjà été montré pour ψ_1 et ψ_2.
($\Rightarrow$) : On montre que pour tous q et k, $v_q[k, \psi] = \top$ implique $q \models \psi$. La preuve est par induction sur k (induction no. 2). Pour $k = 0$, (4.6) et l'hypothèse d'induction no. 1 concluent. Pour $k = k' + 1$, (4.8) donne $q \models \psi_2$ ou bien $q \models \psi_1 \wedge \mathsf{AX}\psi$ (puisque tous les successeurs q' de q vérifient ψ par hypothèse d'induction no. 2). Dans les deux cas, on a $q \models \psi$.
($\Leftarrow$) : Supposons $v_q[n, \psi] = \bot$. Alors $v_q[n, \psi_2] = \bot$ (d'où $q \not\models \psi_2$) et soit $v_q[n, \psi_1] = \bot$ (et alors $v_q \not\models \psi$), soit il existe un successeur q' de q tel que $v_{q'}[n-1, \psi] = \bot$. En itérant cette construction, soit on finit par trouver un chemin issu de q montrant que $q \models \mathsf{E}(\neg\psi_2)\mathsf{U}(\neg\psi_1 \wedge \neg\psi_2)$, soit on trouve un chemin $(q =)q_0, q_1, q_2, \ldots, q_n$ dans S où $q_i \not\models \psi_2$ pour tout i tel que $0 \leq i \leq n$. Dans ce chemin à $n+1$ états, forcément deux états coïncident et donc on peut construire un chemin infini issu de q où tous les états vérifient $\neg\psi_2$. D'où $q \not\models \psi$. □

Preuve (du Théorème 4.6). Cette définition des $v_q[k, \psi]$ peut être vue comme un circuit booléen, tel que celui de la Fig. 4.1, à $O(|S|^2 \times |\varphi|)$ noeuds et dont le noeud $v_q[n, \varphi]$ répond à la question de savoir si $S \models \varphi$. Ce circuit est un dag et on ne peut pas directement l'évaluer en espace logarithmique.

Si on note $v[\psi]$ le vecteur (indexé par Q) des $v_q[n, \psi]$, on peut lire les équations (4.1–4.8) comme des clauses définissant chaque vecteur $v[\psi]$ en fonction de $v[\psi_1]$ et $v[\psi_2]$ où ψ_1 et ψ_2 sont les sous-formules immédiates de ψ. Mémoriser un tel $v[\psi]$ ne demande qu'un espace

$O(|S|)$. Calculer $v[\psi]$ en fonction de $v[\psi_1]$ et $v[\psi_2]$ est une opération simple qui n'a besoin que d'un espace $O(|S|)$ (ceci même pour $\mathsf{E}\psi_1\mathsf{U}\psi_2$ et $\mathsf{A}\psi_1\mathsf{U}\psi_2$ où on doit évaluer itérativement les $v[k,\psi]$ pour $k = 0,\dots,n$).

Par ailleurs, on sait qu'évaluer une expression e (vue comme un arbre) n'a besoin de mémoriser à chaque instant qu'au plus $h(e)$ résultats intermédiaires, où $h(e)$ est le nombre de Strahler de l'arbre sous-jacent à l'expression e [Ers58]. On sait que si e est un arbre binaire à r feuilles, on a $h(e) \leq \log_2 r$. On en déduit que notre calcul n'a besoin que d'un espace $O(|S| \times \log|\varphi|)$. D'où le Théo. 4.6. □

4.2.5 La complexité du model-checking symbolique

Comme nous l'avons déjà mentionné, dans les applications pratiques l'utilisateur du model-checking a surtout besoin d'analyser des structures de Kripke de grande taille plutôt que de vérifier des formules complexes.

Les structures de Kripke modélisent les états internes de systèmes où plusieurs composants interagissent et conduisent à une combinatoire explosive. Ces composants peuvent être des modules communicants entre eux, ou simplement des variables du programme. Chaque composant est facile à comprendre et à visualiser pour le concepteur ou l'utilisateur mais c'est la combinaison de comportements interagissant entre eux qui rend le système difficile à analyser sans la force brute d'un outil automatique.

Une formalisation possible de cette situation utilise les produits d'automates synchronisés à la Arnold et Nivat [AN82, Arn92]. Nous n'allons pas donner une définition formelle ici, et noterons simplement $S_1 \times \cdots \times S_n$ le produit synchronisé des n structures de Kripke. L'essentiel dans cette construction est qu'elle donne lieu à une structure S dont les états sont les n-uplets constitués des états respectifs des S_i et où la « synchronisation » consiste à ne conserver qu'un sous-ensemble des états et des transitions du produit libre selon un critère local (par exemple, $\langle q_1,\dots,q_n\rangle$ n'est conservé dans S que si les $l_i(q_i)$ coïncident sur certaines propositions fixées) [4].

Définition 4.8. (Le problème du model-checking symbolique)
Données d'entrée : *une suite $S_1,\dots,S_n$ de structures de Kripke, ainsi qu'une formule temporelle φ.*
Résultat : « vrai » *si $S \models \varphi$ où $S \stackrel{def}{=} S_1 \times \cdots \times S_n$,* « faux » *sinon.*

Bien sûr, toute la difficulté du model-checking symbolique vient du fait que la description via la suite $S_1,\dots,S_n$ de l'argument S est *implicite* [5] et surtout *très concise*, dans la mesure où $|S|$ est en $O(|S_1| \times \cdots \times |S_n|)$, soit $O(2^{|\text{données d'entrée}|})$.

Un « *algorithme naïf* » pour le model-checking symbolique est un algorithme qui commence par construire explicitement le produit $S = S_1 \times \cdots \times S_n$, et qui ensuite décide si S satisfait

[4] Il existe de nombreuses variantes : structures où les transitions sont étiquetées dans un but de synchronisation, automates agissant sur des variables à domaine fini (par exemple, booléennes), réseaux de Petri *1-safe* où chaque place peut être vue comme une variable booléenne, etc.
La formulation que nous avons retenue a l'avantage de la simplicité, et en un sens tous les modèles évoqués plus haut sont équivalents dans la mesure où chacun permet de coder succinctement les composants et l'opération de synchronisation de chaque autre modèle.

[5] Dans le cas standard où S est donné explicitement, c'est-à-dire en extension, on parle de *méthodes énumératives*.

φ par une méthode énumérative. Dans la plupart des cas pratiques, un outil reposant sur un algorithme naïf ne parviendra pas à simplement achever la construction de S.

Un *algorithme symbolique* pour le model-checking symbolique est un algorithme qui ne représente pas forcément explicitement les états du produit $S = S_1 \times \cdots \times S_n$, et qui peut manipuler des ensembles d'états en passant par des représentations symboliques [6]. Ainsi, ces algorithmes parviennent souvent à analyser des systèmes dont la représentation en extension serait de taille astronomique. Il existe aussi des algorithmes qui ne sont pas « symboliques » sans pour autant être « naïfs », par exemple les algorithmes de vérification à la volée [SW91, DSC99].

Hélas, dans les cas où les résultats théoriques existent, ils montrent que dans certaines situations l'algorithme naïf fait aussi bien que tout autre algorithme :

Théorème 4.9. [KVW00] *Pour LTL, le model-checking symbolique est PSPACE-complet, et sa complexité en programme est PSPACE-complet.*
Pour CTL, le model-checking symbolique est PSPACE-complet, et sa complexité en programme est PSPACE-complet.
Pour CTL^*, le model-checking symbolique est PSPACE-complet, et sa complexité en programme est PSPACE-complet.

4.3 Notes bibliographiques

La preuve que le model-checking de formules LTL sur des systèmes finis est décidable date de [Pnu77]. Sistla et Clarke ont montré qu'il est PSPACE-complet [SC85].

Le premier model-checker pour une logique temporelle [7] est CESAR [QS82] : il évalue des formules de $B(\mathsf{F})$ sur un système de transitions en se basant sur la caractérisation des modalités temporelles comme points-fixes de transformateurs de prédicats.

Le terme « *model-checker* » est dû à [CES86], qui propose un algorithme en $O(|S| \times |\varphi|)$ pour le model-checking de CTL. Leur système EMC prend en compte CTL+équité.

Pour LTL, les algorithmes de model-checking en $O(|S| \times 2^{|\varphi|})$ datent de [LP85] et [VW86b]. Ils sont complexes à implémenter et n'ont réellement été utilisés qu'à partir du développement de Spin [Hol91, Hol97].

L'utilisation de BDDs pour le model-checking symbolique est due à [BCM$^+$92], d'où est issu le model-checker SMV [McM93]. L'approche symbolique de SMV permet d'implémenter facilement le model-checking de LTL [CGH97].

[6] Le plus souvent à base de BDDs [BCM$^+$92, CBM90, CGH97, Rav97].

[7] Il existait déjà des vérificateurs automatiques de systèmes finis qui n'utilisaient pas le formalisme de la logique temporelle, Cf. par exemple [WZ78].

Chapitre 5

Model-checking de *LTL*

Les résultats présentés dans ce chapitre sont tirés de [DS02].

5.1 Motivations

La question à l'origine des travaux décrits dans [DS02] est la suivante : « existe-t-il des fragments naturels de *LTL* pour lesquels le model-checking est plus facile que le PSPACE-complet du Théorème 4.1 ? ».

Découvrir de tels fragments pourrait peut-être fournir une justification formelle à l'opinion souvent exprimée comme quoi les spécifications *LTL* utilisées en pratique sont bien plus simples et régulières que les formules construites pour les preuves de bornes inférieures de complexité.

Par exemple, Dams a introduit récemment le « *until* plat » (une version restreinte de *until*, cf. section 5.6) en expliquant qu'il donnait un pouvoir d'expression suffisant en pratique, et en espérant que la vérification serait plus simple pour des propriétés utilisant *until* plat que pour des propriétés utilisant *until* [Dam99].

Un argument plus fréquemment développé concerne la hauteur temporelle des formules : la majorité des propriétés temporelles utilisées pour spécifier les systèmes ont une hauteur temporelle très limitée.

Il se trouve que, pour la plupart des logiques modales, le problème de la satisfaisabilité devient polynômial quand on ne considère que des formules de hauteur modale bornée (ceci quelque soit la borne) [Hal95].

Inspiré par ces résultats, nous avons entrepris de voir comment la complexité du model-checking (et de la satisfaisabilité) pour *LTL* évoluait quand on se restreignait à des formules de hauteur temporelle bornée et/ou n'utilisant qu'un nombre borné de propositions atomiques.

Définition 5.1. *On note* $\mathsf{L}^k_n(\mathsf{H}_1, \mathsf{H}_2, \ldots)$ *le fragment de LTL où les formules :*

- *sont de hauteur temporelle* $\leq k$ *;*
- *n'utilisent pas plus de* n *propositions distinctes ;*
- *n'utilisent que les combinateurs* $\mathsf{H}_1, \mathsf{H}_2, \ldots$.

$\mathsf{L}^\omega_n(\ldots)$ et $\mathsf{L}^k_\omega(\ldots)$ désignent le fragment sans limitation sur, respectivement, la hauteur temporelle et le nombre de propositions. On écrira $\mathsf{L}(\ldots)$ au lieu de $\mathsf{L}^\omega_\omega(\ldots)$, de sorte que *LTL* est $\mathsf{L}(\mathsf{U},\mathsf{X})$.

Notre étude n'a pas vraiment permis de découvrir des fragments intéressants de LTL pour lesquels le model-checking est plus simple. Elle nous a plutôt permis de mieux comprendre comment différents paramètres interviennent pour provoquer la PSPACE-complétude. Notons toutefois que nos résultats sur le fragment $\mathsf{L}^1_\omega(\mathsf{U},\mathsf{X})$ se sont révélés très utiles pour l'étude des logiques du temps arborescent de la famille de CTL^+ [RS00a, LMS01].

Le reste de cette section présente quelques résultats significatifs, soit par leur énoncé proprement dit, soit par la technique de preuve qu'ils mettent en œuvre.

5.2 Hauteur temporelle bornée pour $\mathsf{L}(\mathsf{U})$

Sistla et Clarke [SC85] ont montré que le model-checking de $\mathsf{L}(\mathsf{U})$ est PSPACE-complet, c.-à-d. que le model-checking de LTL est difficile même sans le combinateur X.

Nous allons montrer que ce résultat est déjà vrai à la hauteur temporelle $k = 2$ (et donc que LTL ne se comporte pas comme les logiques modales de [Hal95]).

5.2.1 Hauteur temporelle $k = 2$

Théorème 5.2. [DS02] *Le model-checking de $\mathsf{L}^2_\omega(\mathsf{U})$ est PSPACE-complet.*

La preuve repose sur une construction réduisant le problème QBF (Quantified Boolean Formula) à un problème de model-checking pour $\mathsf{L}^2_\omega(\mathsf{U})$, construction que nous allons détailler dans le reste de cette section.

Soit P une instance de QBF. Sans perte de généralité, on suppose que P est de la forme

$$P \;\equiv\; \forall x_1\, \exists x_2 \forall x_3 \ldots \exists x_n \overbrace{\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} l_{i,j}}^{P_0} \tag{5.1}$$

où n est un entier pair, et où P_0 est une CNF sans quantificateur, où chaque $l_{i,j}$ est une variable propositionnelle $x_{r(i,j)} \in X$ ou sa négation $\neg x_{r(i,j)}$, avec $X = \{x_1, \ldots, x_n\}$. On note $\epsilon(i,j)$ pour le signe ($+$ ou $-$) du littéral $l_{i,j}$.

Le lemme suivant permet de donner un point de vue plus déclaratif à la notion de validité d'une formule QBF. On l'obtient en reformulant la sémantique des quantificateurs $\forall$ et $\exists$ dans QBF.

Lemme 5.3. *Une formule P de la forme (5.1) est valide ssi il existe un ensemble de valuations $\mathcal{V} \subseteq \{\top,\bot\}^X$ qui soit :*

- ***non-vide :*** *$\mathcal{V} \neq \emptyset$;*
- ***correct :*** *pour tout $v \in \mathcal{V}$, on a $v(P_0) = \top$;*
- ***fermé :*** *pour tout $v \in \mathcal{V}$, pour tout r impair, il existe $v' \in \mathcal{V}$ tel que $v'[x_r] \neq v[x_r]$ et $v'[x_{r'}] = v[x_{r'}]$ pour tout $r' < r$.*

Partant de P, nous construisons la structure de Kripke S_P telle qu'illustrée à la Figure 5.1 où l'ensemble des étiquettes est donné par $Prop \stackrel{\text{def}}{=} \{A_0, A_1, \ldots, x_1^T, \ldots, L_1^1, \ldots\}$. Un calcul π de S_P débute dans l'état s_0. Entre s_0 et s_n, π choisit une valuation des variables de X, atteint

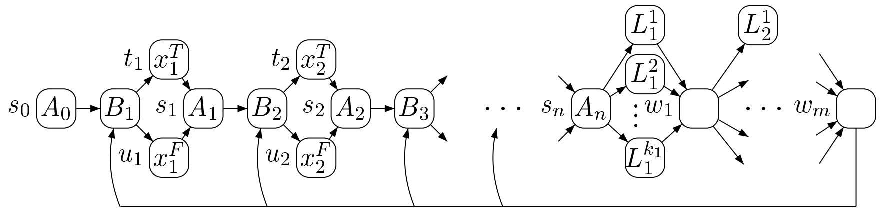

FIG. 5.1 – La structure S_P associée à une instance P de QBF.

finalement w_m et revient sur un état étiqueté par B_r ($1 \leq r \leq n$) où de nouvelles valuations pour $x_r, x_{r+1}, \ldots, x_n$ peuvent être choisies, etc., et ce schéma se répète indéfiniment.

Ainsi, à toute position i où π visite s_n, on peut associer une *valuation courante*, associant à $x_r \in X$ la valeur $\top$ ou $\bot$ selon que c'est u_r ou t_r qui a été visité le dernier. Enfin, à π on associe l'ensemble $\mathcal{V}(\pi)$ de toutes les valuations qui sont courantes à une position i telle que $\pi(i) = s_n$ (il existe une infinité de tels i).

Il est possible d'exprimer certaines propriétés de $\mathcal{V}(\pi)$ via des propriétés temporelles sur π. Pour commencer, définissons les formules

$$\begin{aligned} \psi_r &\stackrel{\text{def}}{=} \mathsf{G}\Big(A_{r-1} \Rightarrow (\neg B_{r-1} \mathsf{U} x_r^T) \wedge (\neg B_{r-1} \mathsf{U} x_r^F)\Big) \qquad \text{pour } 1 \leq r \leq n, \\ \psi_{\text{ferm}} &\stackrel{\text{def}}{=} \psi_1 \wedge \psi_3 \wedge \psi_5 \wedge \cdots \wedge \psi_{n-1}. \end{aligned}$$

Si $\pi \models \psi_{\text{ferm}}$ alors $\mathcal{V}(\pi)$ est fermé (au sens du Lemme 5.3). En effet, soit $v \in \mathcal{V}(\pi)$ et r impair entre 1 et n. Par définition, v est courante à une position i de π. Soit j la plus grande position avant i telle que $\pi(j) = s_{r-1}$. Alors $\pi^j \models A_{r-1}$, et donc $\pi^j \models (\neg B_{r-1} \mathsf{U} x_r^T) \wedge (\neg B_{r-1} \mathsf{U} x_r^F)$, c'est-à-dire que π^j visite u_r et t_r avant de passer par B_{r-1}. On obtient ainsi deux valuations $v_1, v_2 \in \mathcal{V}(\pi)$ qui diffèrent sur x_r mais qui coïncident sur $x_1, \ldots, x_{r-1}$ puisque π doit passer par B_{r-1} pour que la valuation courante soit modifiée sur $x_1, \ldots, x_{r-1}$.

De même, soit $\psi_{\text{corr}} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{k_i} \psi_{i,j}$ où, pour $i = 1, \ldots, m$ et $j = 1, \ldots, k_i$, on a

$$\psi_{i,j} \stackrel{\text{def}}{=} \begin{cases} \mathsf{G}[x_r^F \Rightarrow \mathsf{G}\neg L_i^j \vee \neg L_i^j \mathsf{U} B_r] & \text{si } l_{i,j} = x_r, \\ \mathsf{G}[x_r^T \Rightarrow \mathsf{G}\neg L_i^j \vee \neg L_i^j \mathsf{U} B_r] & \text{si } l_{i,j} = \neg x_r. \end{cases}$$

Si $\pi \models \psi_{\text{corr}}$, alors $\mathcal{V}(\pi)$ est correct (au sens du Lemme 5.3). En effet, soit v la valuation courante en i tel que $\pi(i) = s_n$. Entre les positions i et $i + 2m$ (où π visite w_m) π visite des noeuds étiquetés par $L_1^{a_1}, L_2^{a_2}, \ldots, L_m^{a_m}$. Or $\pi(i + 2p - 1) \models L_p^{a_p}$ implique $v(l_p^{a_p}) = \top$ puisque $\pi \models \psi_{p,a_p}$. D'où l'on déduit que $v(P_0) = \top$.

Finalement, $\pi \models \psi_{\text{ferm}} \wedge \psi_{\text{corr}}$ implique que P est valide.

Il reste à montrer la réciproque. Le plus délicat est de voir que si $\mathcal{V}$ est fermé, alors il existe un chemin π vérifiant ψ_{ferm} et tel que $\mathcal{V}(\pi) = \mathcal{V}$. Un tel chemin est construit en énumérant $\mathcal{V}$ dans un ordre lexicographique $v_1, v_2, \ldots, v_N$. On construit π de sorte que les valuations courantes soient $v_1, v_2, \ldots, v_N, v_1, v_2, \ldots$ dans cet ordre. Entre v_i et v_{i+1}, π va de w_m à B_r où $1 \leq r < n$ est le plus petit indice tel que $v_i(x_r) \neq v_{i+1}(x_r)$. Il est ensuite facile de voir comment modifier π pour garantir $\pi \models \psi_{\text{corr}}$: puisque $v(P) = \top$, les $L_p^{a_p}$ à visiter pour chaque $v \in \mathcal{V}(\pi)$ (c.-à-d. entre s_n et w_m) existent bien.

On a donc le Lemme suivant :

Lemme 5.4. *$S_P, s_0 \models \neg(\psi_{\text{ferm}} \wedge \psi_{\text{corr}})$ ssi P n'est pas valide.*

Ce qui établit la correction de notre réduction, et donc termine la preuve du Théorème 5.2.

5.2.2 Hauteur temporelle $k < 2$

La hauteur $k = 2$ est la valeur minimale pour la PSPACE-complétude de $\mathsf{L}^k_\omega(\mathsf{U})$:

Théorème 5.5. [DS02] *Le model-checking de $\mathsf{L}^1_\omega(\mathsf{U}, \mathsf{X})$ est coNP-complet.*

Le sens « coNP-difficile » est évident : le model-checking pour $\mathsf{L}(\mathsf{F})$ est coNP-difficile et la preuve classique [SC85, Théorème 3.5] n'utilise que des formules de $\mathsf{L}^1_\omega(\mathsf{F})$.

Pour montrer le sens « appartenance à coNP », nous utilisons une propriété des formules de $\mathsf{L}^1_\omega(\mathsf{U}, \mathsf{X})$: ce fragment « a des chemins polynômiaux », comme nous allons l'expliquer maintenant.

Les chemins polynômiaux dont nous parlons sont des chemins *ultimement périodiques* de taille polynomiale. Si $\pi = q_0, q_1, q_2, \ldots$ est un chemin ultimement périodique tel que $q_{i+k'} = q_i$ pour tout $i \geq k$, on définit la taille de π, notée $|\pi|$, comme valant $k + k'$ (on peut en effet représenter π via une liste de $k + k'$ états).

Définition 5.6. *Un fragment L de LTL* a des chemins polynômiaux *si il existe un polynôme p tel que pour toute formule $\varphi \in \mathsf{L}$ et toute structure de Kripke S on a la propriété : si S admet un chemin vérifiant φ alors il existe un tel chemin qui soit ultimement périodique et de taille bornée par $p(|\varphi|, |S|)$.*

Remarque 5.7. Un résultat fondamental de [SC85] énonce que toute formule φ de LTL admet des chemins ultimement périodiques, au sens où pour toute structure S et état q, $S, q \models \mathsf{E}\varphi$ implique l'existence d'un chemin ultimement périodique π partant de q et vérifiant φ. Pour $\varphi \in LTL$ on peut garantir que $|\pi|$ est en $O(2^{|\varphi|} \times |S|)$. ☐

Lemme 5.8. *Si le fragment L de LTL a des chemins polynômiaux (et est clos par négation), le model-checking de L est dans coNP.*

Démonstration. On considère l'algorithme non-déterministe qui, pour S et φ donnés, devine un chemin π ultimement périodique de taille au plus $p(|\neg\varphi|, |S|)$, et qui ensuite vérifie en temps polynômial qu'on a effectivement $\pi \models \neg\varphi$. ☐

On prouve maintenant le sens « appartenance à coNP » du Théorème 5.5 en utilisant le résultat suivant :

Proposition 5.9. *Le fragment $\mathsf{L}^1_\omega(\mathsf{U}, \mathsf{X})$ a des chemins polynômiaux.*

Dans ce mémoire, nous ne donnerons pas la preuve que $\mathsf{L}^1_\omega(\mathsf{U}, \mathsf{X})$ admet des chemins polynômiaux, pour laquelle nous renvoyons à [DS02]. En fait, l'existence de chemins polynômiaux peut être montrée pour $\mathsf{L}^1_\omega(\mathsf{U}, \overset{\infty}{\mathsf{F}}, \mathsf{X})$ [LMS01] et plus généralement pour le fragment de hauteur de quantification au plus 2 de la logique du premier ordre $FO(<, P_A, \ldots)$ [RS00a].

5.3 Hauteur temporelle bornée pour $\mathsf{L}(\mathsf{F}, \mathsf{X})$

Sistla et Clarke [SC85] montrent que le model-checking de $\mathsf{L}(\mathsf{F}, \mathsf{X})$ est PSPACE-complet, c.-à-d. que le model-checking de LTL est difficile même si on remplace le combinateur U par le moins expressif F.

Nous allons montrer que pour tout $k \in \mathbb{N}$, le model-checking est plus simple pour le fragment $\mathsf{L}^k_\omega(\mathsf{F},\mathsf{X})$ que pour $\mathsf{L}(\mathsf{F},\mathsf{X})$:

Théorème 5.10. [DS02] *Pour tout $k > 0$, le model-checking de $\mathsf{L}^k_\omega(\mathsf{F},\mathsf{X})$ est coNP-complet.*

La preuve de ce résultat réduit le model-checking de $\mathsf{L}^k_\omega(\mathsf{F},\mathsf{X})$ au model-checking de $\mathsf{L}^k_\omega(\mathsf{F})$. Puisque le model-checking est coNP-complet pour $\mathsf{L}(\mathsf{F})$ [SC85] et coNP-difficile pour $\mathsf{L}^1_\omega(\mathsf{F})$ (Théorème 5.5), on en déduit le résultat annoncé.

La réduction combine deux techniques simples : l'ensevelissement des X et le dépliage partiel des structures de Kripke (appelé *bulldozing*).

5.3.1 Ensevelissement de X

On dit que *les X sont profonds dans φ* si les seules sous-formules $\mathsf{X}\psi$ de φ sont de la forme $\mathsf{X}\mathsf{X}\cdots\mathsf{X}A$ avec $A \in \mathit{Prop}$.

Ainsi, les occurrences de X dans φ n'ont pas de combinateur booléen ni de combinateur temporel (autre que X) dans leur portée. Par exemple, les X sont profonds dans $\big((\mathsf{F}\mathsf{X}A) \wedge (\mathsf{X}\mathsf{X}\mathsf{X}B)\big)\mathsf{U}C$.

Lemme 5.11. *On peut transformer en temps polynômial toute formule φ de LTL en une formule φ' équivalente et où les X sont profonds.*

Démonstration. Il suffit d'appliquer les équivalences suivantes comme des règles de réécriture (de la gauche vers la droite) :

$$\mathsf{X}\neg\varphi \equiv \neg\mathsf{X}\varphi \qquad \mathsf{X}(\varphi \wedge \psi) \equiv (\mathsf{X}\varphi) \wedge (\mathsf{X}\psi) \qquad \mathsf{X}(\varphi\mathsf{U}\psi) \equiv (\mathsf{X}\varphi)\mathsf{U}(\mathsf{X}\psi)$$

Remarquons que ce procédé garantit que φ' appartient aux mêmes fragments $\mathsf{L}^k_n(\mathsf{H}_1,\ldots)$ que φ. De plus $|\varphi'|$ est en $O(|\varphi|^2)$. ☐

5.3.2 *Bulldozing*

Il ne reste plus qu'à montrer comment coder des X profonds dans les étiquettes des états par une technique de dépliage partiel.

Soient $S = \langle Q, R, l\rangle$ une structure de Kripke, et $k > 0$ un entier. $S^{(k)} = \langle Q^{(k)}, R^{(k)}, l^{(k)}\rangle$ est la structure de Kripke obtenue en dépliant partiellement S. Formellement, les états de $S^{(k)}$ sont les chemins de longueur k dans S :

$$Q^{(k)} \stackrel{\text{def}}{=} \{\langle q_1,\ldots,q_k\rangle \in Q^k \mid \forall 1 \le i < k, (q_i, q_{i+1}) \in R\}$$

Un état $\langle q_1,\ldots,q_k\rangle$ de $S^{(k)}$ représente q_1 où l'on a d'ores et déjà décidé que les $k-1$ prochains états visités seront $q_2, q_3, \ldots, q_k$. C'est ce que reflète la relation $R^{(k)}$:

$$(\langle q_1,\ldots,q_k\rangle, \langle q'_1,\ldots,q'_k\rangle) \in R^{(k)} \stackrel{\text{def}}{\Leftrightarrow} \begin{cases} (q_k, q'_k) \in R, \text{ et} \\ q'_1 = q_2,\ q'_2 = q_3,\ \ldots,\ q'_{k-1} = q_k. \end{cases}$$

Les étiquettes de $S^{(k)}$ permettent de savoir comment chaque composante d'un état de $Q^{(k)}$ est étiqueté dans S. Soit $\mathit{Prop}^{(k)} \stackrel{\text{def}}{=} \mathit{Prop} \times \{1,\ldots,k\}$: pour $A \in \mathit{Prop}$ et $1 \le i \le k$, on écrit A^i plutôt que (A, i). L'étiquetage de $S^{(k)}$ est alors donné par :

$$l^{(k)}(\langle q_1,\ldots,q_k\rangle) \stackrel{\text{def}}{=} \{A^i \mid 1 \le i \le k \text{ et } A \in l(q_i)\}.$$

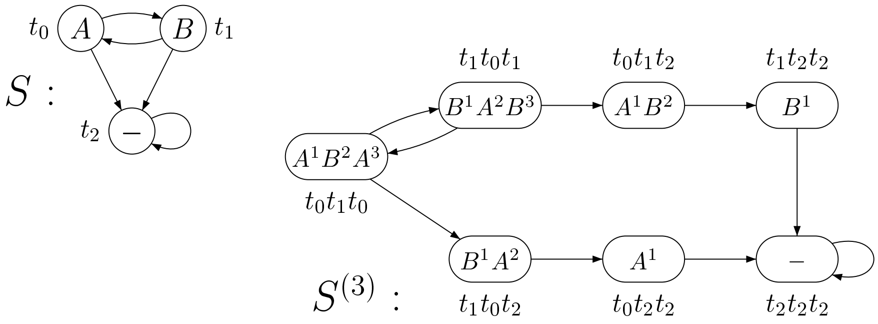

FIG. 5.2 – Un exemple de *bulldozing* : S et $S^{(3)}$.

La Figure 5.2 donne un exemple. Soit maintenant une formule φ de *LTL* où les X sont profonds. φ est de la forme $\psi[\mathsf{X}^{n_1}A_1, \ldots, \mathsf{X}^{n_p}A_p]$ où ψ est sans X (et où $\mathsf{X}^n\varphi$ dénote $\overbrace{\mathsf{X}\cdots\mathsf{X}}^{n}\varphi$).

Soit ψ' la formule obtenue en remplaçant chaque $\mathsf{X}^{n_i}A_i$ dans φ par la proposition $A_i^{1+n_i}$: $\psi' \stackrel{\text{def}}{=} \psi[A_1^{1+n_p}, \ldots, A_p^{1+n_p}]$. Le lemme suivant conclut la réduction.

Lemme 5.12. [DS02] *Si $k > n_i$ pour $i = 1, \ldots, p$, alors $S, q \models \varphi$ ssi $S^{(k)}, \langle q_1, \ldots, q_k\rangle \models \psi'$ pour tout $\langle q_1, \ldots, q_k\rangle \in Q^{(k)}$ tel que $q_1 = q$.*

Remarquons que $|S^{(k)}|$ est en $O(|S|^k)$: la transformation n'est succincte que pour k fixé.

Il est alors facile de combiner les lemmes 5.11 et 5.12 pour prouver le sens « appartenance à coNP » du Théorème 5.10. Remarquons que la réduction obtenue n'est pas encore une *many-one reduction* puisque une instance (S, φ) donne lieu à de nombreuses instances $(S^{(k)}, \varphi'')$. Mais ceci se répare aisément en ajoutant un nouvel état initial à $S^{(k)}$ et en modifiant légèrement φ''.

5.4 Nombre de propositions borné

En général, imposer une borne au nombre de propositions autorisées ne simplifie pas le model-checking. C'est ce que précisent les résultats suivants.

Théorème 5.13. [DS02] *Le model-checking de $\mathsf{L}_2^\omega(\mathsf{U})$ et celui de $\mathsf{L}_1^\omega(\mathsf{F}, \mathsf{X})$ sont PSPACE-complets.*

La preuve (que nous ne présentons pas dans ce mémoire) passe par une réduction du model-checking de $\mathsf{L}(\mathsf{U})$ au model-checking de $\mathsf{L}_2^\omega(\mathsf{U})$, et une réduction du model-checking de $\mathsf{L}(\mathsf{F}, \mathsf{X})$ au model-checking de $\mathsf{L}_1^\omega(\mathsf{F}, \mathsf{X})$. À chaque fois, il s'agit de coder les propositions sur un nombre fixé et limité de propositions distinctes.

Les bornes du Théorème 5.13 sont strictes et on peut montrer :

Théorème 5.14. [DS02] *Le model-checking de $\mathsf{L}_1^\omega(\mathsf{U})$ est coNP-complet.*

Pour prouver ce résultat, nous allons montrer que $\mathsf{L}_1^\omega(\mathsf{U})$ a des chemins polynômiaux.

Nous allons utiliser un résultat du à [Lam83] : si deux chemins π et π' sont *stuttering*-équivalents, ils vérifient les mêmes formules de $\mathsf{L}(\mathsf{U})$ (c'est-à-dire, les mêmes formules sans X).

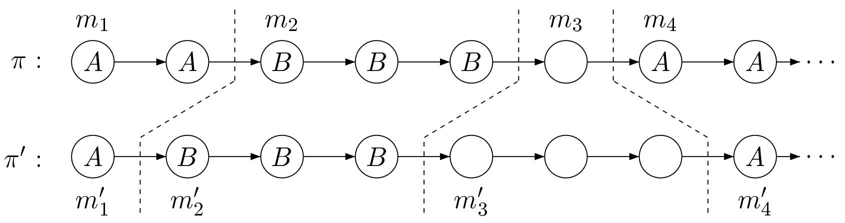

FIG. 5.3 – Deux chemins *stuttering*-équivalents.

Rappelons que, comme illustré par la Figure 5.3, deux chemins π et π' sont *stuttering*-équivalents si il existe un découpage $0 = m_1 < m_2 < m_3 \ldots$ de π et un découpage $0 = m'_1 < m'_2 < m'_3 \ldots$ de π' tel que $l(\pi(i)) = l(\pi'(i'))$ pour tous i et i' dans des tranches correspondantes, c.-à-d. tels que $m_n \leq i < m_{n+1}$ et $m'_n \leq i' < m'_{n+1}$ pour un $n \in \mathbb{N}$.

Les formules de $L_1^\omega(\mathsf{U})$ n'utilisent qu'une seule proposition atomique, par exemple A. Quand *Prop* ne contient que la proposition A, la *stuttering*-équivalence laisse peu de choix : tout chemin dans une structure de Kripke S est équivalent, modulo *stuttering*-équivalence, à un chemin d'une des six formes suivantes :

$$\begin{array}{lll@{\qquad}lll}
\pi_{1,n} & \stackrel{\text{def}}{=} & (A.\neg A)^n.A^\omega\,; & \pi_{4,n} & \stackrel{\text{def}}{=} & (\neg A.A)^n.\neg A^\omega\,; \\
\pi_{2,n} & \stackrel{\text{def}}{=} & \neg A.(A.\neg A)^n A^\omega\,; & \pi_{5,n} & \stackrel{\text{def}}{=} & A.(\neg A.A)^n \neg A^\omega\,; \\
\pi_{3,n} & \stackrel{\text{def}}{=} & (A.\neg A)^\omega\,; & \pi_{6,n} & \stackrel{\text{def}}{=} & (\neg A.A)^\omega.
\end{array}$$

La forme des chemins $\pi_{3,n}$ et $\pi_{6,n}$ ne dépend pas de n, mais la notation $\pi_{k,n}$ permet d'énoncer le lemme suivant :

Lemme 5.15. [DS02] *Si $n \geq h(\varphi)$ pour une formule $\varphi \in LTL$, alors $\pi_{k,n} \models \varphi$ ssi $\pi_{k,n+1} \models \varphi$.*

Démonstration. Par induction structurelle sur φ, et en remarquant que le suffixe $(\pi_{k,n})^1$ d'un chemin $\pi_{k,n}$ est de la forme $\pi_{k',n'}$ avec $n - 1 \leq n' \leq n$. □

Corollaire 5.16. $\mathsf{L}_1^\omega(\mathsf{U})$ *admet des chemins polynômiaux.*

Démonstration. Supposons $S, q \models \mathsf{E}\varphi$ pour $\varphi \in \mathsf{L}_1^\omega(\mathsf{U})$. Il existe donc un chemin ultimement périodique π partant de q et vérifiant φ. Ce chemin π est *stuttering*-équivalent à un $\pi_{k,n}$. Si $n > |S|$ on peut, en éliminant une boucle, extraire de π un chemin π' partant de q et *stuttering*-équivalent à $\pi_{k,n'}$ pour un $n' < n$. Tant que $n' \geq h(\varphi)$ on sait que $\pi' \models \varphi$ par le Lemme 5.15. En itérant cette construction, on construit un π_{k,n^*} montrant $S, q \models \mathsf{E}\varphi$ avec n^* en $O(\max(|S|, h(\varphi)))$. Donc π_{k,n^*} est de taille polynomiale. □

5.5 D'autres fragments remarquables

Le tableau de la Figure 5.4, tiré de [DS02], recense la complexité du model-checking pour tous les fragments $\mathsf{L}_n^k(\mathsf{H}_1, \mathsf{H}_2, \ldots)$ où les combinateurs $\mathsf{H}_1, \ldots$ sont parmi $\mathsf{U}, \mathsf{X}, \mathsf{F}$.

	$n-1, k < \omega$	Model-checking
$\mathsf{L}(\ldots)$	$\mathsf{L}^0_\omega(\ldots)$	dans LOGSPACE
$\mathsf{L}(\mathsf{F})$	$\mathsf{L}(\mathsf{F})$	coNP-complet
	$\mathsf{L}^1_\omega(\mathsf{F})$	coNP-complet
	$\mathsf{L}^\omega_2(\mathsf{F})$	coNP-complet
	$\mathsf{L}^\omega_1(\mathsf{F})$	dans P, NLOGSPACE-difficile
	$\mathsf{L}^{k+1}_n(\mathsf{F})$	NLOGSPACE-complet
$\mathsf{L}(\mathsf{U})$	$\mathsf{L}(\mathsf{U})$	PSPACE-complet
	$\mathsf{L}^2_\omega(\mathsf{U})$	PSPACE-complet
	$\mathsf{L}^1_\omega(\mathsf{U})$	coNP-complet
	$\mathsf{L}^\omega_2(\mathsf{U})$	PSPACE-complet
	$\mathsf{L}^\omega_1(\mathsf{U})$	dans P, NLOGSPACE-difficile
	$\mathsf{L}^{1+k}_n(\mathsf{U})$	NLOGSPACE-complet
$\mathsf{L}(\mathsf{X})$	$\mathsf{L}(\mathsf{X})$	coNP-complet
	$\mathsf{L}^k_\omega(\mathsf{X})$	dans LOGSPACE
	$\mathsf{L}^\omega_1(\mathsf{X})$	coNP-complet
$\mathsf{L}(\mathsf{F},\mathsf{X})$	$\mathsf{L}(\mathsf{F},\mathsf{X})$	PSPACE-complet
	$\mathsf{L}^{2+k}_\omega(\mathsf{F},\mathsf{X})$	coNP-complet
	$\mathsf{L}^1_\omega(\mathsf{F},\mathsf{X})$	coNP-complet
	$\mathsf{L}^\omega_1(\mathsf{F},\mathsf{X})$	PSPACE-complet
	$\mathsf{L}^{1+k}_n(\mathsf{F},\mathsf{X})$	NLOGSPACE-complet
$\mathsf{L}(\mathsf{U},\mathsf{X})$	$\mathsf{L}(\mathsf{U},\mathsf{X})$	PSPACE-complet
	$\mathsf{L}^2_\omega(\mathsf{U},\mathsf{X})$	PSPACE-complet
	$\mathsf{L}^1_\omega(\mathsf{U},\mathsf{X})$	coNP-complet
	$\mathsf{L}^\omega_1(\mathsf{U},\mathsf{X})$	PSPACE-complet
	$\mathsf{L}^{1+k}_n(\mathsf{U},\mathsf{X})$	NLOGSPACE-complet

FIG. 5.4 – Complexité du model-checking pour les fragments bornés de *LTL*.

Les seuls fragments identifiés pour lesquels la complexité du model-checking est en temps polynômial (ou plus bas) sont très particuliers. Ils ne correspondent pas aux types de formule couramment utilisés en pratique : trop peu de propositions, ou hauteur temporelle trop limitée, ou jeu de combinateurs trop réduit.

Les résultats concernant les fragments pour lesquels la complexité reste *intractable* mais n'est plus PSPACE-complète sont très intéressants pour étudier la complexité d'autres logiques.

Nous allons conclure cette section en présentant les idées permettant de comprendre certains des résultats de complexité polynomiale (ou moindre). Il s'agit des Théorèmes 5.17 et 5.18.

Théorème 5.17. [DS02] *Le model-checking de* $\mathsf{L}^0_\omega(\ldots)$ *est dans LOGSPACE.*

Démonstration. Le fragment considéré ne contient que des expressions booléennes sans combinateur temporel. Le model-checking demande donc d'évaluer la valeur de vérité d'une formule booléenne étant donnée une valuation (celle de l'état courant). On peut utiliser l'algorithme classique de [Lyn77]. □

Théorème 5.18. [DS02] *Pour tous* $n, k < \omega$ *tels que* $k > 0$*, le model-checking de* $\mathsf{L}^k_n(\mathsf{U}, \mathsf{X})$ *est NLOGSPACE-complet.*

Démonstration. Le sens « NLOGSPACE-difficile » est une simple réduction de GAP, le problème d'atteignabilité dans un graphe, bien connu pour être NLOGSPACE-complet [Jon75].

Pour montrer le sens « appartenance à NLOGSPACE », remarquons que le fragment considéré, où n et k sont bornés, ne contient qu'un nombre fini [1] N de formules non-équivalentes deux à deux. Notons $\psi_1, \ldots, \psi_N$ les représentants canoniques des formules de $\mathsf{L}^k_n(\mathsf{U}, \mathsf{X})$.

Nous donnons un algorithme montrant que le problème est dans NLOGSPACE. Soient S et φ les données du problème. D'abord il existe un algorithme simple pour calculer le représentant canonique de chaque formule : il suffit de disposer d'une table (finie) pour chaque combinateur temporel disant comment se combinent les ψ_i's. Par exemple, la table pour U contiendra des entrées telles que « $\psi_a \mathsf{U} \psi_b \mapsto \psi_c$ ». En procédant de façon compositionnelle, on peut calculer en espace logarithmique (à nouveau via les techniques de [Lyn77]) le représentant ψ_i de la formule $\neg\varphi$.

Ensuite on lit dans une table quel est l'automate de Büchi $\mathcal{A}_i$ associé à ψ_i. Puisqu'il n'y a qu'un nombre fini N de tels automates, leur taille est en $O(1)$. Il reste à décider si S a un chemin accepté par $\mathcal{A}_i$. Il suffit de construire un produit synchronisé $S \otimes \mathcal{A}_i$ pour se ramener à GAP, et puisque $|\mathcal{A}_i|$ est en $O(1)$, $|S \otimes \mathcal{A}_i|$ est en $O(|S|)$. □

5.6 *Until* plat

Le *until* plat, que nous noterons U^-, est une version de *until* où l'on interdit d'utiliser des combinateurs temporels dans la partie gauche du U. Cette contrainte est respectée dans la formule $A\mathsf{U}^-(B\mathsf{U}^-C)$ mais pas dans la formule $(A\mathsf{U}B)\mathsf{U}C$ car la sous-formule $A\mathsf{U}B$ contient des combinateurs temporels et se trouve dans la partie gauche du U le plus englobant.

C'est Dams [Dam99] qui a proposé cette restriction et a montré formellement que $\mathsf{L}(\mathsf{U}^-)$ est moins expressif que $\mathsf{L}(\mathsf{U})$, même si en pratique la plupart des exemples de spécification

[1] Pour $k = 0$, il y a $N^0_n = 2^{2^n}$ façons différentes de combiner n propositions. Pour $k > 0$ on peut combiner N^{k-1}_n sous-formules, pour un total de N^k_n résultats, avec N^k_n en $2^{2^{O((N^{k-1}_n)^2)}}$.

n'utilisent que U^-.

Dams espérait que le model-checking serait plus simple pour des propriétés utilisant *until* plat que pour des propriétés utilisant *until*.

Notre étude ne laisse guère de place à cet espoir. Par exemple la réduction de QBF dans le model-checking pour $\mathsf{L}^2_\omega(\mathsf{U})$ (Lemme 5.4) utilise une formule $\psi_{\text{ferm}} \wedge \psi_{\text{corr}}$ dont tous les *until*s sont plats. On peut donc directement généraliser le Théorème 5.2 au fragment $\mathsf{L}^2_\omega(\mathsf{U}^-)$.

En fait, pour tous les fragments recensés dans le tableau de la Figure 5.4, [DS02] établit que la complexité est la même quand on remplace U par U^-.

5.7 Notes bibliographiques

Dans le cadre du model-checking temporel, il existe peu de travaux dont la démarche consiste à chercher des fragments *tractables* de problèmes complexes, ceci alors que cette démarche est très répandue dans d'autres domaines.

Mentionnons toutefois [BK98] qui cherche à comprendre pourquoi en pratique `Mona` ne se heurte pas systématiquement à l'explosion combinatoire. Ce travail isole un fragment de la logique WS1S pour lequel la complexité est diminuée.

C'est une motivation semblable qui est à l'origine de [KV98] où les auteurs étudient quelles formules *LTL* ont un équivalent en *CTL*. Un des objectifs est de comprendre pourquoi et comment le model-checking de *LTL* est faisable en pratique.

La restriction au *until* plat est utilisée (sans être nommée) dans [MC85] où elle permet d'appliquer des abstractions de système qui ne seraient pas correctes pour U. Il apparaît aussi dans les logiques pour les modèles du vrai parallélisme [DG99]. De façon remarquable, le *until* plat correspond à une notion indépendante d'automates plats [CC00].

Notons enfin que les fragments $\mathsf{L}(\mathsf{X})$, où X est le seul combinateur, sont considérées comme relevant de la logique modale plutôt que de la logique temporelle (mais voir [SR99] pour un autre point de vue).

Chapitre 6

Model-checking de *PCTL*

Les résultats présentés dans ce chapitre sont tirés de [LS00a].

6.1 Motivations

Nous avons proposé la logique *PCTL* au chapitre 2 pour améliorer l'expressivité des logiques temporelles utilisées pour la spécification. Une utilisation dans une optique de vérification demanderait, par exemple, de disposer d'algorithmes de model-checking pour *PCTL*.

Le model-checking de *PCTL* soulève deux questions :

1. La première question est d'en prouver la décidabilité. En effet, la sémantique de *PCTL* est définie en termes de chemins finis dans la structure de Kripke considérée (cf. Définition 2.2), de sorte qu'il n'est pas évident de voir comment adapter les méthodes énumératives habituelles des logiques pur-futur.
2. La seconde question est de mesurer la complexité intrinsèque du problème et d'en trouver les paramètres pertinents, ceci avec l'idée de justifier qu'un algorithme proposé serait « optimal ».

Nous abordons brièvement la première question dans la section 6.2 et traitons plus profondément la seconde question dans la section 6.3.

6.2 Model-checking de *PCTL* en espace polynômial

Comme pour la plupart des logiques temporelles, la décidabilité du model-checking de *PCTL* peut se déduire de résultats généraux sur la théorie des arbres avec quantifications sur les branches [Gur85, GS85].

Par ailleurs, la décidabilité du model-checking de CTL_{lp}, c.-à-d. « *PCTL* sans N », a été montrée par Kupferman, Pnueli et Vardi (cf. [KP95]) en utilisant des techniques d'automates d'arbres infinis. Ils aboutissent à un algorithme en espace polynômial :

Théorème 6.1. [KP95] *Le model-checking de CTL_{lp} est PSPACE-complet.*

Dans ce mémoire nous allons admettre ce résultat et nous contenter de montrer comment notre problème s'y réduit simplement :

Proposition 6.2. [LS00a] *Le model-checking de PCTL est dans PSPACE.*

Démonstration. Soit φ une formule de *PCTL* et $S = \langle Q, R, l\rangle$ une structure de Kripke. On raisonne par induction sur le nombre k d'occurrences de N dans φ. Si $k = 0$, alors φ est une formule de CTL_{lp}. Sinon on écrit φ sous la forme $\varphi[\mathsf{N}\psi]$ où ψ est sans N. Puisque pour la logique *PCTL* les formules $\mathsf{N}\psi$ sont des formules d'état (c'est-à-dire que $\pi, n \models \mathsf{N}\psi$ ssi $\pi(n) \models \mathsf{N}\psi$ ssi $\pi(n) \models \psi$) on peut les remplacer par des propositions $A_{\mathsf{N}\psi}$ et étiqueter chaque état de S par l' définie par :

$$\forall q \in Q,\quad l'(q) \stackrel{\text{def}}{=} \begin{cases} l(q) \cup \{A_{\mathsf{N}\psi}\} & \text{si } q \models \psi, \\ l(q) & \text{sinon.} \end{cases}$$

Soit $S' = \langle Q, R, l'\rangle$. Pour tout $q \in Q$, on a $q \models \varphi$ dans S ssi $q \models \varphi[A_{\mathsf{N}\psi}]$ dans S'. Or $\varphi[A_{\mathsf{N}\psi}]$ ne contient plus que $k-1$ occurrences de N. Il ne reste plus qu'à continuer dans la même voie, ceci jusqu'à avoir retiré toutes les occurrences de N.

On vient de décrire un algorithme en temps polynômial utilisant un oracle PSPACE (le model-checker de CTL_{lp}, utilisé pour construire l'), c'est-à-dire en fin de compte un algorithme PSPACE, au vu de $\mathrm{P}^{\mathrm{PSPACE}} = \mathrm{PSPACE}$. □

En ce qui concerne CTL^*_{lp} (et donc $PCTL^*$) le problème du model-checking est, à notre connaissance, encore ouvert (même si la décidabilité découle du Théorème 2.11).

6.3 Bornes inférieures pour le model-checking de *PCTL*

Pour prouver le Théorème 6.1, [KP95] montre que le model-checking de CTL_{lp} est PSPACE-difficile. *A fortiori* le model-checking de *PCTL* est PSPACE-difficile.

En fait, on peut montrer simplement que le model-checking de CTL+Passé est PSPACE-difficile dès qu'on s'autorise un peu de passé et un peu de futur dans une logique du temps arborescent.

Soit P une instance du problème QBF (Quantified Boolean Formula). On reprend les notations de la section 5.2 et on suppose que P est de la forme

$$P \;\equiv\; \forall x_1\, \exists x_2 \forall x_3 \ldots \exists x_n \overbrace{\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} l_{i,j}}^{P_0}$$

où n est un entier pair, et où chaque $l_{i,j}$ est une variable propositionnelle $x_{r(i,j)} \in X$ ou sa négation $\neg x_{r(i,j)}$, avec $X = \{x_1, \ldots, x_n\}$. On note $\epsilon(i,j)$ pour le signe ($+$ ou $-$) du littéral $l_{i,j}$.

À partir de P nous construisons une structure de Kripke S_P telle qu'illustrée Figure 6.1 où l'ensemble des étiquettes est donné par $Prop \stackrel{\text{def}}{=} \{x_1^+, x_1^-, \ldots, A_0, \ldots, L_1^1, \ldots\}$. Un calcul π de S_P débute dans q_0, visite q_n et termine sur q_*. On associe à un tel π la valuation v_π des variables propositionnelles $x_1, \ldots, x_n$ définie canoniquement par $v(x_i) \stackrel{\text{def}}{=} \top$ (resp. $\stackrel{\text{def}}{=} \bot$) ssi $x_i^+ \in l_{S_P}(\pi(2i-1))$ (resp. $x_i^- \in l_{S_P}(\pi(2i-1))$). On dit que v_π valide P_0 quand, pour chaque clause $\bigvee_j l_{i,j}$ de P_0, v_π donne la valeur $\top$ à au moins un littéral $l_{i,j}$.

Cette propriété de v_π s'exprime aussi comme une propriété temporelle de π. Soit

$$\varphi_{ok} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{k_i} \Big(L_i^j \Rightarrow \mathsf{F}^{-1} x_{r(i,j)}^{\epsilon(i,j)} \Big).$$

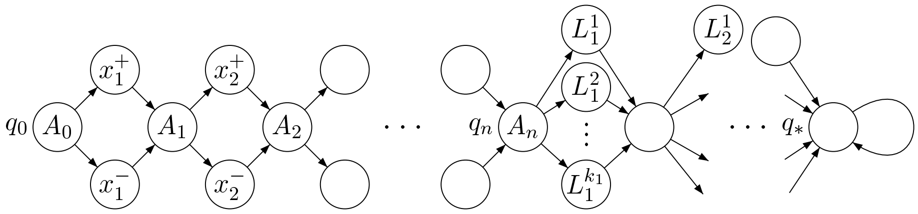

FIG. 6.1 – La structure S_P associée à une instance P de QBF.

Alors v_π valide P_0 ssi $\pi(0), \ldots, \pi(2n) \models \mathsf{EG}\,\varphi_{ok}$, de sorte que P est valide ssi $S_P, q_0 \models \varphi_P$, où

$$\varphi_P \stackrel{\text{def}}{=} \overbrace{\mathsf{AF}\,(A_1}^{\forall x_1} \wedge \overbrace{\mathsf{EF}\,(A_2}^{\exists x_2} \wedge \overbrace{\mathsf{AF}\,(A_3}^{\forall x_3} \wedge \cdots \overbrace{\mathsf{EF}\,(A_n}^{\exists x_n} \wedge \mathsf{EG}\,\varphi_{ok})\cdots))).$$

φ_P est une formule de $B_r(\mathsf{F}^{-1}, \mathsf{F})$, c'est-à-dire le fragment de *PCTL* où F^{-1} et F sont les seuls combinateurs temporels autorisés (pas de N, X, X^{-1}, U ou S) et où l'emboîtement passé-futur est *restreint* (pas de combinateur du futur dans la portée d'un combinateur du passé).

Corollaire 6.3. [LS00a] *Le model-checking de $B_r(\mathsf{F}^{-1}, \mathsf{F})$ est PSPACE-difficile.*

La même structure S_P s'adapte à $B_r(\mathsf{F}^{-1}, \mathsf{X})$. Soit φ'_P la formule suivante :

$$\overbrace{\mathsf{AX}\,\mathsf{AX}}^{\forall x_1}\overbrace{\mathsf{EX}\,\mathsf{EX}}^{\exists x_2}\overbrace{\mathsf{AX}\,\mathsf{AX}}^{\forall x_3}\cdots\overbrace{\mathsf{EX}\,\mathsf{EX}}^{\exists x_n}\mathsf{EX}(\overbrace{\varphi_{ok} \wedge \mathsf{EXEX}(\varphi_{ok} \wedge \mathsf{EXEX}(\cdots\varphi_{ok})}^{m \text{ fois } \varphi_{ok}}\cdots)).$$

Alors P est valide ssi $S_P, q_0 \models \varphi'_P$.

De la même façon, on peut remplacer F^{-1} dans φ_{ok} par des X^{-1} emboîtés :

$$\varphi'_{ok} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m}\bigwedge_{j=1}^{k_i}\Big(L_i^j \Rightarrow \mathsf{X}^{-2i-2m+2r(i,j)}x_{r(i,j)}^{\epsilon(i,j)}\Big).$$

En utilisant φ'_{ok} au lieu de φ_{ok} dans φ_P et φ'_P, on réduit P à une instance du model-checking de $B_r(\mathsf{F}, \mathsf{X}^{-1})$ et $B_r(\mathsf{X}, \mathsf{X}^{-1})$.

Théorème 6.4. [LS00a] *Le model-checking de $B_r(\mathsf{F}^{-1}, \mathsf{F})$, de $B_r(\mathsf{F}^{-1}, \mathsf{X})$, de $B_r(\mathsf{X}^{-1}, \mathsf{F})$ ou de $B_r(\mathsf{X}^{-1}, \mathsf{X})$ est PSPACE-complet.*

6.4 Notes bibliographiques

Le théorème de séparation pour *LTL*+Passé [GPSS80, Gab89] implique implicitement la décidabilité du model-checking pour cette logique (et pour un fragment de *CTL**+Passé, cf. [HT87]) mais ceci sans en mesurer la complexité [1].

La littérature contient des algorithmes pour la satisfaisabilité de logiques avec passé, algorithmes qui pourraient être utilisés pour le model-checking : [Var88, RMD+92] construisent des automates de Büchi pour le μ-calcul linéaire passé+futur et, respectivement, *LTL*+Passé. [Var98] considère le μ-calcul arborescent passé+futur.

[1] De fait on ne sait même pas si la procédure de séparation de *LTL*+Passé est non-élémentaire comme on le soupçonne.

Des model-checkers ont été réalisés pour LTL+Passé : [KMMP93] décrit un système construisant incrémentalement l'automate de Büchi et [Rav97] utilise la technique des variables d'histoire pour se ramener au cas pur-futur et invoquer un model-checker symbolique classique.

Chapitre 7

Model-checking de $TCTL$

Les résultats présentés dans ce chapitre sont tirés de [LST02].

7.1 Motivations

Dans le chapitre 3, nous avons montré comment les logiques $TCTL_{\mathrm{s}}$ et $TCTL_{\mathrm{q}}$ étendent CTL et permettent d'exprimer certaines propriétés quantitatives de façon très succincte.

Ici nous cherchons à savoir quand et comment cette expressivité accrue se traduit par une augmentation du coût du model-checking.

7.2 Model-checking de $TCTL_{\mathrm{q}}$

Concernant $TCTL_{\mathrm{q}}$, les nouvelles sont plutôt mauvaises :

Lemme 7.1. [LST02] *Le model-checking de $TCTL_{\mathrm{q}}$ est PSPACE-difficile, même dans le cas d'une structure de Kripke fixée.*

Démonstration. On réduit une instance P de QBF (Quantified Boolean Formula) en un problème de model-checking $S \models \Phi_P$ pour $TCTL_{\mathrm{q}}$. Pour que S ne dépende pas de P, on s'inspire d'une technique de [AL99].

Sans perte de généralité, P est de la forme $\mathsf{Q}_1 p_1 \dots \mathsf{Q}_n p_n . \varphi$ avec $\mathsf{Q}_i \in \{\exists, \forall\}$ et φ une formule booléenne sur les atomes $p_1, \dots, p_n$. S est la structure $S \stackrel{\text{def}}{=} \langle \{q\}, \{q \to q\}, l : q \mapsto \{tick\} \rangle$ et Φ_P est la formule

$$t \;\mathtt{in}\; \mathsf{EF}\Big[t{=}1 \wedge \mathsf{EO}_1 \Big(x_1 \;\mathtt{in}\; \mathsf{EF}\Big[t{=}2 \wedge \dots \Big(t{=}i \wedge \mathsf{EO}_i (x_i \;\mathtt{in}\; \mathsf{EF}(t{=}i{+}1 \wedge \dots$$
$$\mathsf{EF}(t{=}n{+}1 \wedge \tilde{\varphi}) \dots \Big) \Big] \Big) \Big]$$

où chaque O_i est soit $\mathsf{F}_{\leq 1}$ soit $\mathsf{G}_{\leq 1}$ suivant que Q_i est $\exists$ ou $\forall$, et où $\tilde{\varphi}$ est obtenu en remplaçant chaque p_i par $x_i = n + 1 - i$ dans φ.

La seule exécution de S est $qq\dots$, et puisque chaque horloge x_i est introduite dans Φ_P entre $\mathsf{EF}(t = i \wedge \cdots)$ et $\mathsf{EF}(t = i + 1 \wedge \cdots)$, elle ne peut être initialisée qu'à la date i ou $i + 1$.

Soit v une valuation de $\{t, x_1, \dots, x_n\}$, et i une valuation de t, on a :

$$q, v \models \mathsf{EF}_{\leq 1}\Big(x_i \;\texttt{in}\; \mathsf{EF}(t{=}i{+}1 \wedge \psi)\Big) \quad \text{ssi} \quad \begin{cases} q, (v[x_i/0]) + 1 \models \psi \\ \textbf{ou } q, (v+1)[x_i/0] \models \psi. \end{cases}$$

$$q, v \models \mathsf{EG}_{\leq 1}\Big(x_i \;\texttt{in}\; \mathsf{EF}(t{=}i{+}1 \wedge \psi)\Big) \quad \text{ssi} \quad \begin{cases} q, (v[x_i/0]) + 1 \models \psi \\ \textbf{et } q, (v+1)[x_i/0] \models \psi. \end{cases}$$

Ainsi Φ_P est une transcription exacte de P et $S \models \Phi_P$ ssi P est vraie. □

Cette borne inférieure est optimale :

Lemme 7.2. *Il existe un algorithme pour le model-checking de $TCTL_{\mathrm{q}}$ qui n'utilise que $O(|S| \times |\varphi|_{\mathrm{g}}^2)$ espace mémoire.*

Démonstration. Une preuve simple passe par la traduction du model-checking de $TCTL_{\mathrm{q}}$ en un problème de model-checking symbolique pour CTL.

En effet, la question « $S, v \models \varphi$? » peut s'exprimer sous la forme « $S \times S_v \models \varphi'$? » où S_v est une structure de Kripke qui se comporte comme l'ensemble des valuations dans la définition 3.8 et où φ' est maintenant une formule CTL.

Si φ utilise les horloges $x_1, \dots, x_n$, les états de S_v sont les points de $\mathbb{N}^n$ mais le Lemme 3.4 permet de restreindre S_v à un système fini de taille $M_\varphi^{x_1} \times \cdots \times M_\varphi^{x_n}$ (où M_φ^x désigne la plus grande constante à laquelle x est comparée dans φ), qui est en fait un produit synchronisé $S_v^{x_1} \times \cdots \times S_v^{x_n}$. Chaque S_v^x a M_φ^x états mais on peut l'exprimer comme un produit de $\lceil \log M_\varphi^x \rceil$ structures à deux états.

En fin de compte, on a traduit « $S \models \varphi$? » en un problème de model-checking symbolique « $S \times S_1 \times \cdots \times S_k \models \varphi'$? » où $|\varphi'|_{\mathrm{g}} = O(|\varphi|_{\mathrm{g}})$, $|S_i| = O(1)$ et $k = \sum_{i=1}^n \lceil \log M_\varphi^{x_i} \rceil = O(|\varphi|_{\mathrm{g}})$. □

Théorème 7.3. [LST02]
Le model-checking de $TCTL_{\mathrm{q}}$ est PSPACE-complet ; de plus sa complexité en programme est NLOGSPACE-complet, sa complexité en formule est PSPACE-complet.
Le model-checking symbolique de $TCTL_{\mathrm{q}}$ est PSPACE-complet ; de plus sa complexité en programme est PSPACE-complète.

7.3 Model-checking de $TCTL_{\mathrm{s}}$

Le résultat remarquable concernant le model-checking de $TCTL_{\mathrm{s}}$ est qu'il n'est pas plus coûteux que le model-checking de CTL, alors même que $TCTL_{\mathrm{s}}$ peut être exponentiellement plus concise que CTL (Lemme 3.4).

Théorème 7.4. [LST02]
On peut effectuer le model-checking de $TCTL_{\mathrm{s}}$ en temps $O(|S|^3 \times |\varphi|)$, et en temps $O(|S| \times |\varphi|)$ si on se limite à $TCTL_{\mathrm{s}}^{<,>}$.

Idée de la démonstration. L'algorithme qui permet d'obtenir ces résultats étend l'algorithme classique pour CTL. Nous illustrons l'idée sur un exemple : supposons que nous voulions marquer tous les états de S qui vérifient $\mathsf{E}A\mathsf{U}_{<k}B$. Il suffit de modifier l'algorithme classique pour $\mathsf{E}A\mathsf{U}B$ en visitant les états selon un parcours en largeur, de sorte que pour chaque état q vérifiant $\mathsf{E}A\mathsf{U}B$ on calcule le n minimum tel que $q \models \mathsf{E}A\mathsf{U}_{<n}B$. Ce parcours se fait en temps $O(|S|)$ et on peut procéder de même pour $\mathsf{A}A\mathsf{U}_{<k}B$.

Pour $\mathsf{E}A\mathsf{U}_{>k}B$ on commence à calculer quels états sont dans une composante fortement connexe vérifiant $\mathsf{EG}A$ et contenant un *tick*. On peut marquer les états vérifiant $\mathsf{E}A\mathsf{U}B$ en mémorisant pour chaque q quel est le n maximum tel que $q \models \mathsf{E}A\mathsf{U}_{>n}B$. Évidemment, un état d'une des composantes connexes évoquées plus haut reçoit $n = \infty$. Ce parcours se fait en temps $O(|S|)$ et on peut procéder de même pour $\mathsf{A}A\mathsf{U}_{>k}B$.

Pour $\mathsf{E}A\mathsf{U}_{=k}B$ il faut une procédure plus compliquée. L'idée ici est de calculer la relation $R_k \stackrel{\text{def}}{=} \{(q, q') \mid q \models \mathsf{E}A\mathsf{U}_{=k}q'\}$. R_k peut se calculer par induction sur la décomposition binaire de k. Par exemple, R_{2n} est $R_n \circ R_n$ et R_{2n+1} se calcule de façon similaire. Finalement, on peut construire R_k en temps $O(|Q|^3 \times \lceil \log k \rceil)$. $\square$

Théorème 7.5. [LST02]
Le model-checking de $TCTL_{\mathrm{s}}$ est P-complet ; de plus sa complexité en programme est NLOGSPACE-complet, sa complexité en formule est LOGSPACE.
Le model-checking symbolique de $TCTL_{\mathrm{s}}$ est PSPACE-complet ; de plus sa complexité en programme est PSPACE-complet.

7.4 Notes bibliographiques

Quand on interprète des logiques telles que *TCTL* sur des automates temporisés, les algorithmes de model-checking reposent sur la technique assez complexe du graphe des régions [ACD93]. Le problème est PSPACE-complet.

La complexité bilinéaire du model-checking de $TCTL_{\mathrm{s}}^{<,>}$ généralise le résultat de [EMSS92] pour une logique semblable à $TCTL_{\mathrm{s}}^{<}$.

La complexité du model-checking sort facilement du cadre polynômial dès qu'on ajoute certains types de contraintes. Par exemple, le model-checking de la logique *RTCTL+*, où l'on peut utiliser des expressions régulières comme contraintes de temporisation, est PSPACE-complet [ET97]. Quand on autorise la coexistence de différentes échelles de temps, le model-checking de $\mathsf{E}A\mathsf{U}_{\dots}B$ est NP-complet [ET99]. Quant à la logique de [YMW97], son problème de model-checking est indécidable.

Troisième partie

Vérification des systèmes infinis

Introduction

C'est assez récemment que la vérification des systèmes infinis a été identifiée comme un domaine de recherche en tant que tel, même si de nombreux résultats sont anciens.

Par *système infini* on entend tout système dont le nombre de configurations n'est pas nécessairement borné [1]. Il s'agit aussi bien de réseaux de Petri, d'automates à files (ou à piles, ou à compteurs), de processus d'une algèbre à la CCS, etc.

Pour ces systèmes, on s'intéresse à toutes les questions classiques de vérification (atteignabilité, terminaison, model-checking en logique temporelle, équivalence de deux systèmes). L'objectif est d'étendre la démarche du model-checking à des systèmes dans lesquels la modélisation est plus naturelle. Pour cette raison, les systèmes étudiés sont en général moins puissants que les machines de Turing [2]. Les résultats de ces dernières années proposent une cartographie (encore très incomplète) de quelles propriétés sont décidables pour quels systèmes sous quelles hypothèses.

Aujourd'hui il est difficile de présenter de façon organisée et synthétique les frontières de décidabilité telles qu'elles ont été révélées par les chercheurs. Nous avons préféré axer notre présentation sur deux travaux qui illustrent chacun une technique originale pour les résultats de décidabilité [3].

Les problèmes de bases que nous considérons ne sont plus uniquement des questions de logique temporelle, aussi nous allons rappeler (dans le chapitre 8) quels sont les principaux problèmes étudiés, et quels liens existent entre eux. Nous en profiterons pour donner en passant les notations utilisées pour les systèmes de transitions étiquetés.

Les deux sections suivantes sont consacrées à deux modèles particuliers pour lesquels nous voulons mettre en avant des contributions originales : ce sont les systèmes bien structurés (Chapitre 9) et les processus PA (Chapitre 10).

[1] Par contraste avec les structures de Kripke de la partie II, et les produits de structures de Kripke de la Définition 4.8.

[2] Il s'agit parfois de versions dégradées des modèles les plus puissants : Lossy Channel Systems [Fin94, CFP96, AJ96b, AJ96a], machines à compteurs avec contrôle plat [CC00], CCS avec parallélisme mais sans synchronisation [CHM94]. Les réseaux de Petri peuvent être vus comme des machines à compteurs avec un test à zéro incomplet.

[3] Les résultats d'indécidabilité utilisent eux des méthodes plus variées et se prêtent moins à la classification. Par ailleurs ils présentent en général moins d'intérêt et se justifient surtout vis-à-vis d'un résultat de décidabilité correspondant à un modèle ou une propriété un peu moins général(e).

Chapitre 8

Les problèmes de base

Les divers systèmes que nous considérerons (leur définition formelle sera donnée dans les chapitres concernés) partagent certaines caractéristiques. Ce sont tous des descriptions finies d'un espace de configurations $S = \{c, c', \ldots\}$ potentiellement infini, ainsi que de règles permettant d'engendrer des transitions $c \xrightarrow{a} c'$ (où $a \in Act$ est un *nom d'action*). Une configuration est souvent composée d'un *état de contrôle* $q \in Q$ (où Q est fini) et d'une valuation σ qui contient les valeurs courantes des variables.

8.1 Les problèmes d'atteignabilité

Les relations $c \xrightarrow{a} c'$ entre configurations peuvent être enchaînées et donnent naturellement lieu à une relation $c \xrightarrow{w} c'$ pour $w \in Act^*$. On note $c \xrightarrow{*} c'$ (resp. $c \xrightarrow{+} c'$) pour $\exists w \in Act^*, c \xrightarrow{w} c'$ (resp. $\exists w \in Act^+ \cdots$).

Le problème de l'*atteignabilité* [1] (d'une configuration) consiste à décider si $c \xrightarrow{*} c'$ pour c, c' donnés.

Le problème de l'*atteignabilité d'un état de contrôle* consiste à décider, étant donnés une configuration c et un état de contrôle $q \in Q$, si il existe une valuation σ telle que $c \xrightarrow{*} c' = (q, \sigma)$, c.-à-d. si on peut atteindre une configuration où l'état de contrôle est q.

Ce problème est une variante du problème de l'atteignabilité mais il est beaucoup plus utile en pratique : la vérification des propriétés de sûreté se ramène presque toujours à une question de non-atteignabilité d'un état de contrôle.

Dans de nombreux modèles (les réseaux de Petri, ...) les deux problèmes se réduisent facilement l'un dans l'autre ce qui explique pourquoi certains travaux ne les distinguent pas clairement. En ce qui concerne les systèmes infinis les plus couramment étudiés, on constate que l'atteignabilité d'un état de contrôle est souvent un problème plus simple que l'atteignabilité d'une configuration. C'est le cas p.ex. de tous les systèmes bien structurés (Cf. chapitre 9).

Enfin, le problème de l'*atteignabilité répétée* (d'un état de contrôle) consiste à décider si, étant donnés une configuration c et un état de contrôle $q \in Q$, il existe une exécution « passant par q » une infinité de fois, c.-à-d. s'il existe une suite infinie $\sigma_1, \sigma_2, \ldots$ de valuations telle que

[1] Aussi appelée *accessibilité*.

$c \xrightarrow{*} (q, \sigma_1)$ et, pour $i = 1, 2, \ldots$, $(q, \sigma_i) \xrightarrow{+} (q, \sigma_{i+1})$.

L'atteignabilité répétée est une propriété de vivacité et, pour la plupart des systèmes [2], elle permet d'exprimer toutes les propriétés *LTL*.

En général l'atteignabilité répétée est plus difficile que la simple atteignabilité : pour de nombreux systèmes [3] l'atteignabilité répétée est indécidable tandis que les deux autres problèmes d'atteignabilité sont décidables.

8.2 Le caractère borné

Le problème du *caractère borné* (« boundedness » en anglais) est celui de la finitude de l'ensemble d'atteignabilité, pour une configuration de départ donnée.

Le problème du caractère borné n'est pas seulement théorique : c'est un problème de vérification qui se pose en pratique dans les situations où un modèle d'états infinis est utilisé pour modéliser un système qui doit nécessairement être borné, p.ex. pour certains protocoles communicant par canaux fifo [JJ93].

Le caractère borné est un problème beaucoup moins populaire que l'atteignabilité ou la terminaison. C'est néanmoins un problème fondamental dans l'analyse des systèmes infinis, ceci pour deux raisons :

- Le caractère borné permet de savoir si le système infini examiné est réellement infini, non analysable par les méthodes énumératives classiques.
- Les deux principales mesures de coût des programmes sont le temps et l'espace mémoire requis. Le caractère borné est le pendant pour l'espace mémoire de la terminaison pour le temps.

Un système non-borné ne termine pas. Un système borné ne termine pas forcément. C'est le caractère borné, et non la terminaison, qui est la vraie frontière entre les systèmes que l'on peut analyser trivialement et ceux pour lesquels les méthodes énumératives ne suffisent plus.

8.3 Les équivalences et les préordres

Une des plus anciennes notions d'équivalence entre systèmes est l'égalité des langages engendrés (ou acceptés). De fait, tout l'enseignement de base de la théorie des langages formels adopte implicitement le point de vue selon lequel deux automates acceptant le même langage sont interchangeables. Dans cette optique, décider de l'équivalence de deux systèmes est un problème fondamental qui n'a pas besoin de justification.

Mais l'équivalence de deux systèmes est aussi un *problème de vérification*. Dans les situations où l'on voit un des deux systèmes comme la spécification abstraite S_{spec} d'un comportement et l'autre système comme une modélisation S_{impl} de son implémentation, on dira que S_{impl} est une implémentation correcte de S_{spec} si les deux systèmes sont équivalents.

Cette démarche est très répandue, au moins quand l'équivalence en question fait abstraction de certains détails inutiles (p.ex., elle cache les actions de S_{impl} qui doivent être invisibles quand on se met au niveau de S_{spec}), et de nombreux outils de vérification (tels qu'Alde-

[2] Il suffit que la famille de systèmes considérée soit close pour la synchronisation avec un automate fini.

[3] Par exemple, les Lossy Channel Systems [AJ96a]. Les réseaux de Petri sont une exception intéressante, pour lesquels on peut réduire les questions d'équité à des questions d'atteignabilité [Jan90].

baran [Fer89], Auto [BRDV90] et la Concurrency Workbench [CPS93]) résolvent ce type de questions sur les systèmes finis.

8.4 Les problèmes de model-checking temporel

Les problèmes de model-checking temporel regroupent toutes les questions de la forme « est-ce que $S, c \models \varphi$? » pour une classe de formules temporelles.

Les liens entre le model-checking temporel et l'atteignabilité sont nombreux. D'une part, les problèmes d'atteignabilité sont des problèmes temporels particuliers. Dans l'autre direction, le model-checking de logiques du temps linéaire telles que LTL se réduisent à des questions d'atteignabilité répétée (pour toutes les classes de modèles clos par intersection avec un automate fini). De même, le model-checking de certaines logiques du temps arborescent telles que CTL, peut s'aborder comme un calcul de prédécesseurs itérés.

Dans le cas des systèmes infinis, il est fréquent que le model-checking de logiques du temps arborescent soit indécidable, et qu'on ne parvienne à montrer la décidabilité que pour des fragments réduits tels que UB, voire EF ou EG, deux logiques qui ne comportent qu'un seul combinateur temporel, respectivement EF et EG.

Notons enfin que certains résultats de décidabilité pour les systèmes infinis concernent des logiques qui ne sont pas à proprement parler temporelles (par exemple la logique monadique du second-ordre sur le graphe des configurations) mais qui elles aussi généralisent les questions d'atteignabilité.

8.5 Notes bibliographiques

8.5.1 Équivalences et préordres

Pour les systèmes infinis, le problème de la vérification de l'équivalence a longtemps été considéré comme non-pertinent, dans la mesure où l'équivalence de langages est indécidable pour la quasi-totalité de ces systèmes : tout le monde sait que l'équivalence des grammaires hors-contextes, ou des automates à pile, est indécidable. Le même résultat négatif vaut pour les réseaux de Petri [Hac76].

Le problème de l'équivalence est devenu pertinent quand on s'est rendu compte que l'équivalence de bisimulation est décidable pour certains systèmes infinis bien choisis [4] [BBK93]. Il suffit donc de considèrer la bisimulation comme l'équivalence sémantique canonique [5] qui servira de critère pour la relation d'implémentation correcte. Un tel point de vue présente beaucoup d'intérêt : la bisimulation est réellement un critère d'équivalence plus simple et plus fondamental que l'égalité des langages, mieux en accord avec l'utilisation de logiques temporelles et avec le raisonnement modulaire sur les systèmes [BK89, Mil89, Gla01].

8.5.2 Model-checking temporel

Les systèmes infinis pour lesquels des résultats positifs existent sont des modèles où l'atteignabilité est décidable. Les principales familles sont les réseaux de Petri [HR89, HRY91,

[4] Par exemple, la bisimulation est décidable sur les processus PA normés [HJ99] (voir les notes de la section 10.6 pour plus de références sur cette approche) et les automates à un compteur [Jan00] mais pas sur les réseaux de Petri [Jan95].

[5] De fait, quasiment toutes les autres équivalences sont indécidables [GH94].

Jan90, Esp97a], les automates à pile et *pushdown processes* [MS85, HS93, BS94, BEM97]. Il existe d'autres modèles, p.ex. [May01, BM99].

8.5.3 Variantes

Les résultats de décidabilité sur les systèmes infinis s'expriment parfois en termes de variantes des problèmes que nous avons mentionnés dans cette introduction :

Atteignabilité : il existe d'autres notions d'atteignabilité (modulo une équivalence [May97c], etc.) dont nous verrons un exemple section 10.5.

Caractère borné : la régularité au sens de [Kuč99] est une variante sémantique de la notion de caractère borné.

Équivalences et préordres : d'autres notions d'équivalences sont décidables pour certains systèmes infinis, mais il s'agit souvent de variantes de la bisimulation, p.ex. la similarité ou la bisimilarité avec un système fini [AK95, JM95, JK97, JEM99, JKM01, KM02].

Notons enfin que pour les systèmes de transitions *déterministes*, la décidabilité de l'équivalence des langages (comme dans [Sén00] par exemple) peut être vue comme une décidabilité de la bisimulation puisque, dans ce cadre, la bisimulation coïncide avec l'égalité des langages (et avec toutes les équivalences du spectre de van Glabbeek [Gla01]).

Chapitre 9

L'analyse des systèmes bien structurés

Les résultats présentés dans ce chapitre sont tirés de [FS98, FS01].

9.1 Motivations

Les systèmes bien structurés (ou « WSTS », de l'anglais *well-structured systems*) forment une famille de systèmes infinis pour lesquels existent des résultats généraux de décidabilité. Il s'agit de systèmes où l'existence d'un beau préordre compatible avec les transitions garantit certaines propriétés sémantiques (p.ex., le Lemme 9.5), ainsi que la finitude ou la terminaison de certaines constructions (p.ex., les Lemmes 9.1 et 9.2). De façon élégante, les algorithmes proposés pour l'analyse des WSTS tirent parti de ces deux sortes de propriétés.

Dans ce chapitre nous présentons les notions de base de la théorie des WSTS. Nous renvoyons à la section 9.7 pour plus de détails sur l'origine de cette théorie, et à la deuxième partie de [FS01] pour une liste non-exhaustive de WSTS tirés des domaines suivants : réseaux de Petri étendus, algèbres de processus, systèmes de réécriture et grammaires formelles, automates interprétés, etc.

9.2 Beaux préordres

Rappelons qu'un beau préordre (« well-quasi-ordering », ou *wqo*, en anglais) est un préordre $(X, \leq)$ tel que de toute suite infinie $s_1, s_2, \ldots$ on peut extraire une paire $s_i \leq s_j$ avec $i < j$ [Kru72] (un wqo est donc bien fondé).

En informatique, on rencontre de nombreux exemples de wqo : $\mathbb{N}$ avec l'ordre naturel, $\mathbb{N}^k$ avec l'ordre produit (Lemme de Dickson [Dic13]), Σ^* et Σ^ω avec l'ordre de sous-mot (Lemme de Higman [Hig52, Fin85]), T_Σ (l'ensemble des termes sur la signature Σ) avec l'ordre de plongement (Théorème de Kruskal [Kru60, NW63]).

Soit $(X, \leq)$ un wqo. Un sous-ensemble $Y \subseteq X$ est *fermé vers le haut* si $x \geq y$ et $y \in Y$ entraînent $x \in Y$. Une *base* de Y est une partie Y' de Y telle que $Y = \uparrow Y'$, où $\uparrow Y'$ dénote la fermeture (vers le haut) de Y' :

$$\uparrow Y' \stackrel{\text{def}}{=} \{x \in X \mid \exists y \in Y', y \leq x\}.$$

Les deux principales propriétés que nous utiliserons sont classiques :

Lemme 9.1. *Tout ensemble $Y \subseteq X$ fermé vers le haut admet une base finie.*

Lemme 9.2. *Si $Y_1 \subseteq Y_2 \subseteq \cdots$ est une suite infinie croissante de fermés vers le haut, alors il existe un rang fini k à partir duquel la suite se stabilise : $Y_k = Y_{k+1} = Y_{k+2} = \ldots$.*

9.3 Systèmes bien structurés

Après ces préliminaires, il est possible de donner la définition fondamentale :

Définition 9.3. [FS01] *Un* système bien structuré *(WSTS) est un système de transitions $\langle S, \rightarrow\rangle$ muni d'un wqo $\leq \subseteq S \times S$* compatible *avec les transitions, c.-à-d. tel que pour tous états c_1, c_1', c_2 de S :*

$$\textit{si } c_1 \rightarrow c_2 \textit{ et } c_1 \leq c_1' \textit{ alors il existe } c_2' \geq c_2 \textit{ tel que } c_1' \xrightarrow{*} c_2'.$$

Par rapport aux propositions antérieures (Cf. section 9.7), la définition 9.3 introduit deux nouveautés : (1) on relègue à plus tard toute hypothèse sur l'effectivité du wqo (ou de la relation $\rightarrow$), et (2) on autorise une suite de transitions $t \xrightarrow{*} t'$ au lieu d'exiger un unique pas $t \rightarrow t'$. Ce sont des considérations méthodologiques qui nous ont amené à ne faire figurer les hypothèses d'effectivité que dans les énoncés des algorithmes sur les WSTS. Notons toutefois la conséquence surprenante :

Théorème 9.4. [FS01] *Tout système de transitions $\langle S, \rightarrow\rangle$ peut être muni d'un wqo qui le rend bien structuré.*

La seconde nouveauté est moins fondamentale [1], mais elle souligne un défaut des définitions antérieures qui conduisaient à restreindre artificiellement la granularité des transitions des modèles étudiés, de façon à les faire rentrer dans le moule de la définition. P.ex., [CFP96, AJ96b] doivent considérer qu'une transition « atomique » d'un *Lossy Channel System* se compose d'un échange de messages précédé et suivi d'un nombre arbitraire de pertes de messages, alors que la Définition 9.3 permet de considérer qu'une transition atomique est soit un échange soit une (unique) perte. Ce genre de distinctions a parfois des conséquences importantes, par exemple quand la propriété à vérifier dépend de la granularité des transitions comme c'est le cas pour l'inévitabilité dans le Théorème 9.12.

Dans un WSTS $\langle S, \rightarrow, \leq\rangle$, la compatibilité entre le wqo $\leq$ et les transitions $\rightarrow$ peut se lire en termes plus sémantiques : $\leq$ est compatible avec $\rightarrow$ ssi $\leq$ est une simulation pour $\xrightarrow{*}$. On en déduit la conséquence suivante :

Lemme 9.5. *Si $Y \subseteq S$ est fermé vers le haut, alors $Pred^*(Y)$ est fermé vers le haut.*

Démonstration. Supposons $c_1 \geq c_2 \in Pred^*(Y)$. Alors il existe un $c_2' \in Y$ tel que $c_2 \xrightarrow{*} c_2'$, et la propriété de simulation entraîne l'existence d'une suite $c_1 \xrightarrow{*} c_1' \geq c_2'$. Or $c_1' \in Y$ car Y est fermé vers le haut, et donc $c_1 \in Pred^*(Y)$. □

9.4 Décidabilité de la recouvrabilité

Le problème de la recouvrabilité (d'une configuration) consiste à décider, étant données deux configurations $c, c' \in S$ si on peut couvrir c' depuis c, c.-à-d. si il existe $c'' \geq c'$ tel que $c \xrightarrow{*} c''$.

[1] Et moins nouvelle, cf. [Fin90].

« Couvrir » signifie donc ici atteindre une configuration plus grande dans l'ordre $\leq$ du WSTS, et dépend donc de cet ordre. En pratique, on peut considérer que la recouvrabilité est une variante de l'atteignabilité d'un état de contrôle (p.ex., il y a coïncidence exacte pour les Lossy Channel Systems) pour des modèles où il n'existe pas forcément d'états de contrôle.

Sous certaines hypothèses générales, la recouvrabilité est décidable pour les WSTS :

Proposition 9.6. [FS01] *Soit $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ un WSTS tel que :*
(1) le wqo $\leq$ est décidable ;
(2) la fonction predmin, où predmin(c) est l'ensemble des prédécesseurs minimaux de c, est calculable [2].
Alors il est possible de calculer, étant donné $c \in S$, l'ensemble des éléments minimaux de $Pred^*(\uparrow\{c\})$.

Idée de la démonstration. La preuve repose sur deux propriétés clefs : (1) $Pred^*(\uparrow\{c\})$ est fermé vers le haut (Lemme 9.5), et (2) le calcul itéré $Y_0 \stackrel{\text{def}}{=} \{c\}$, $Y_{i+1} \stackrel{\text{def}}{=} Y_i \cup predmin(Y_i)$ converge, au sens où il doit exister un rang k tel que $\uparrow Y_k = \uparrow Y_{k+1}$ (Lemme 9.2). $\square$

On peut ainsi calculer l'ensemble des c à partir desquels une configuration c' est recouvrable, donc *a fortiori* on peut affirmer :

Théorème 9.7. [FS01] *La recouvrabilité est décidable pour un WSTS $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ où $\leq$ et predmin sont effectifs.*

Dans les modèles où $c_1 \geq c_2$ implique $c_1 \xrightarrow{*} c_2$ (p.ex. les Lossy Channel Systems), recouvrabilité et atteignabilité coïncident et le Théorème 9.7 donne la décidabilité de l'atteignabilité.

Dans les réseaux de Petri, la décidabilité de la recouvrabilité était connue depuis [KM69]. Il est remarquable de noter que la procédure sous-jacente au Théorème 9.7 (une procédure de chaînage arrière) est beaucoup plus simple à décrire, à justifier, et à implémenter que la construction de l'arbre ou du graphe de couverture [KM69, Fin93], seule méthode connue pendant 25 ans (une procédure de chaînage avant). Qui plus est, la procédure en arrière s'applique aussi à de nombreuses extensions des réseaux de Petri [DFS98, DJS99, LS00c, May00c] pour lesquelles on sait qu'il n'existe pas d'arbres de couverture [EFM99].

9.5 Décidabilité des propriétés d'inévitabilité

La Définition 9.3 est très générale, et il existe des notions plus fortes de compatibilité entre $\leq$ et $\rightarrow$.

Définition 9.8. *Un WSTS $\langle S, \rightarrow, \leq \rangle$ a la propriété de*

compatibilité forte *si $c_1 \rightarrow c_2$ et $c_1 \leq c'_1$ impliquent qu'il existe $c'_2 \geq c_2$ tel que $c'_1 \rightarrow c'_2$;*

compatibilité réflexive *si $c_1 \rightarrow c_2$ et $c_1 \leq c'_1$ impliquent qu'il existe $c'_2 \geq c_2$ tel que $c'_1 \xrightarrow{=} c'_2$ (c.-à-d. $c'_1 \rightarrow c'_2$ ou $c'_1 = c'_2$) ;*

compatibilité transitive *si $c_1 \rightarrow c_2$ et $c_1 \leq c'_1$ impliquent qu'il existe $c'_2 \geq c_2$ tel que $c'_1 \xrightarrow{+} c'_2$;*

[2]L'hypothèse de l'effectivité de *predmin* peut sembler lourde et complexe, mais on peut la comprendre en imaginant que, moralement, elle revient à dire qu'on peut construire l'ensemble des prédécesseurs sans exiger que cet ensemble soit fini.

compatibilité de branchement [3] *si* $c_1 \to c_2$ *et* $c_1 \leq c'_1$ *impliquent qu'il existe une suite de transitions* $c'_1 \to c'_2 \to c'_3 \dots c'_n$ *(*$n = 1$ *est possible) telle que* $c'_n \geq c_2$ *et* $c'_i \geq c_1$ *pour tout* $1 \leq i < n$.

Pour les WSTS avec compatibilité transitive, on peut décider certaines propriétés d'inévitabilité au moyen de l'arbre d'atteignabilité :

Définition 9.9. [Fin90] *L'arbre d'atteignabilité depuis* c*, noté* $FRT(c)$*, est l'arbre étiqueté construit avec la procédure suivante :*

- *la racine de l'arbre est étiquetée par* c *;*
- *un noeud* n_1 *étiqueté par* c_1 *est* mort *si il existe un noeud* $n_2 \neq n_1$*, ancêtre de* n_1*, et étiqueté par* c_2 *tel que* $c_2 \leq c_1$ *(et sinon* n_1 *est* vivant*) ;*
- *un noeud mort est une feuille de l'arbre ;*
- *pour chaque successeur immédiat* c' *d'une configuration* c *étiquetant un noeud vivant* n*, on ajoute un fils* n' *de* n *étiqueté par* c'*.*

Ainsi, $FRT(c)$ est l'arbre des comportements issus de c, modulo le fait que la construction est interrompue pour les configurations c_1 dont on a déjà rencontré un minorant c_2 situé entre la racine et c_1. Les feuilles sont les noeuds morts et les noeuds étiquetés par des configurations sans successeur.

Si $\mathcal{S}$ est un WSTS à compatibilité de branchement, alors $FRT(c)$ a des propriétés sémantiques. Soit $Y \subseteq S$ un ensemble fermé vers le bas :

Lemme 9.10. *Les exécutions de* $\mathcal{S}$ *partant de* c *rencontrent inévitablement un état de* Y *ssi les chemins de* $FRT(c)$ *rencontrent inévitablement un état de* Y*.*

Démonstration. Le sens « $\Leftarrow$ » est évident puisque les (étiquettes des) chemins de $FRT(c)$ sont des préfixes des exécutions de $\mathcal{S}$. Pour « $\Rightarrow$ » on raisonne par l'absurde et on suppose qu'un chemin de $FRT(c)$, de la forme $n_0 : c_0 \to \dots \to n_k : c_k$ (avec $c_0 = c$) ne rencontre pas Y. Si c_k n'a pas de successeur, alors le chemin est une exécution de $\mathcal{S}$. Sinon n_k est un noeud mort, et $c_l \leq c_k$ pour un $l < k$. À partir de $c_l \leq c_k$ et $c_l \xrightarrow{+} c_k$ on construit itérativement, grâce à la compatibilité de branchement, une exécution infinie $\pi = c_0 \to \dots c_k \to c_{k+1} \dots$ où pour tout $i \geq k$ il existe un j, $l \leq j < k$ tel que $c_i \geq c_j$. Alors, puisque les c_i pour $i \leq k$ sont dans $S - Y$, et puisque $S - Y$ est fermé vers le haut, toutes les configurations de π sont dans $S - Y$. $\square$

$FRT(c)$ est donc une représentation fidèle de $\mathcal{S}$ pour des propriétés de la forme $c \models \mathsf{AF}Y$ où Y est fermé vers le bas. Or $FRT(c)$ est plus facile à construire que $\mathcal{S}$.

Premièrement, puisque $\leq$ est un wqo, le Lemme de König donne immédiatement :

Proposition 9.11. *Si* $\mathcal{S}$ *est à branchement fini alors* $FRT(c)$ *est fini.*

Si de plus $\leq$ est effectif, et si on peut calculer l'ensemble des successeurs immédiats de toute configuration, alors $FRT(c)$ est effectivement calculable.

On en déduit comme corollaire :

Théorème 9.12. [FS01] *Les problèmes de la forme* $c \models \mathsf{AF}\neg\uparrow\{c_1, \dots, c_n\}$ *sont décidables pour un WSTS à branchement fini, avec compatibilité de branchement, et où* $\leq$ *et Post sont effectifs.*

[3] Traduction proposée pour la « stuttering compatibility » de [FS01].

9.6 D'autres formes de compatibilité et d'autres résultats de décidabilité

[FS01] propose d'autres formes de compatibilité :

la compatibilité *stricte* où c'est l'ordre strict $<$ qui est une relation de simulation, comme par exemple dans les réseaux de Petri ;

la compatibilité *vers le bas* où c'est l'ordre inverse $\geq$ qui est une relation de simulation (mais c'est $\leq$ qui est wqo), comme par exemple dans les BPP avec arc inhibiteurs ;

la compatibilité *en arrière* où $\leq$ est compatible avec $\rightarrow^{-1}$, comme par exemple dans les systèmes communicants avec erreurs d'insertion de [CFP96].

Ces différentes formes de compatibilité se combinent et ne sont pas forcément indépendantes, p.ex. les réseaux de Petri classiques avec l'ordre canonique d'inclusion entre les marquages donnent lieu à des WSTS pour la compatibilité stricte ou non, transitive ou de branchement ou forte, en avant ou en arrière.

Les différentes formes de compatibilité sont les hypothèses minimales correspondant à différents résultats de décidabilité. Citons par exemple les deux théorèmes suivants (pour lesquels nous omettons d'énoncer les hypothèses d'effectivité nécessaires) :

Théorème 9.13. [FS01] *La finitude de l'ensemble d'atteignabilité est décidable pour les WSTS avec compatibilité transitive stricte.*

Théorème 9.14. [FS01] *Il est possible de calculer une base de* $\uparrow Post^*(c)$ *pour les WSTS avec compatibilité réflexive vers le bas.*

9.7 Notes bibliographiques

Les WSTS ont été introduits par Finkel [Fin87, Fin90], puis redécouverts et développés par [AČJT00]. Ils forment la classe STS5 de [HM00].

Indépendamment de ces travaux, l'étude des schémas RPP m'avait conduit à prouver la décidabilité de la terminaison (et d'autres problèmes d'inévitabilité) de ces schémas par la même technique que celle du Théorème 9.12 [KS97b, KS97a]. De même, [KS97b, Théorème 5.3] a donné naissance à la notion de compatibilité « vers le bas » et anticipe le Théorème 9.14.

Par la suite, A. Finkel et moi-même avons proposé une définition plus générale des WSTS, ce qui a permis d'appliquer la théorie à un grand nombre de modèles tirés de la littérature sur la vérification des systèmes infinis.

On peut mesurer l'intérêt de l'idée des systèmes bien structurés au nombre de travaux récents qui l'utilisent ou bien en poursuivent l'exploration [Jan99, AN00, DR00, FPS00, Kus00, AN01].

Chapitre 10

L'analyse des processus PA

Les résultats présentés dans ce chapitre sont tirés de [LS02, LS00d].

10.1 Motivations

PA est une des plus riches algèbres de processus pour lesquelles des résultats de décidabilité existent.

L'étude de telles algèbres est un domaine actif depuis une dizaine d'années. On peut voir ces algèbres comme des abstractions (ou schémas) de programmes récursifs et les utiliser pour l'analyse statique, comme dans [EK99, EP00].

Les particularités de ce domaine de recherche sont (1) les modèles s'inspirent des systèmes de réécriture plutôt que des automates ; (2) les problèmes étudiés sont surtout des questions d'équivalence (p.ex. par bisimulation) ; (3) les techniques algébriques héritées de CCS jouent un rôle important. Nous renvoyons à la section 10.6 pour des notes historiques sur le sujet.

Dans ce chapitre nous montrons une approche simple et nouvelle, à base d'automates d'arbres, pour les questions d'atteignabilité dans PA.

10.2 L'algèbre PA

10.2.1 Syntaxe

On considère un alphabet fini $Act = \{a, b, \ldots\}$ d'étiquettes (ou noms d'action) et on note Act^* l'ensemble des mots sur Act, où ϵ est le mot vide. Étant donnés deux mots $w_1, w_2 \in Act^*$, $w_1 \sqcup\!\sqcup\, w_2$ dénote leur mélange (ou « shuffle product »).

Soit un ensemble $Const = \{X, Y, Z, \ldots\}$ de *noms de processus* (ou *constantes*), l'ensemble $E_{\mathrm{PA}} = \{s, t, \ldots\}$ des termes, ou processus [1], PA est donné par la grammaire suivante

$$s, t ::= s.t \mid s \parallel t \mid 0 \mid X \mid Y \mid Z \mid \cdots$$

Pour $t \in E_{\mathrm{PA}}$, on note $Const(t)$ l'ensemble des noms de processus apparaissant dans t.

[1] En général, on parle d'un *terme PA* quand on s'intéresse à sa structure, et d'un *processus PA* quand on s'intéresse à son comportement.

Une *déclaration* PA est un ensemble fini $\Delta \subseteq Const \times Act \times E_{\text{PA}}$ de *règles* (de réécriture). Une règle $(X, a, t) \in \Delta$ est notée $X \xrightarrow{a} t$.

Pour un terme $t \in E_{\text{PA}}$, on note $Subterms(t)$ l'ensemble des sous-termes de t. De même $Subterms(\Delta)$ est l'ensemble des sous-termes apparaissant dans (un terme de) Δ. La taille d'un terme (ou d'une déclaration) est donnée par

$$|t| \stackrel{\text{def}}{=} Card(Subterms(t)), \qquad\qquad |\Delta| \stackrel{\text{def}}{=} Card(Subterms(\Delta)).$$

10.2.2 Sémantique

La déclaration Δ définit une relation de transition entre les processus E_{PA} et transforme E_{PA} en un système de transitions étiqueté $(E_{\text{PA}}, \to)$ où $\to \subseteq E_{\text{PA}} \times Act^* \times E_{\text{PA}}$. La relation est définie par les règles SOS [2] suivantes :

$$(\mathrm{R}_\epsilon)\quad \frac{}{0 \xrightarrow{\epsilon} 0} \qquad\qquad (\mathrm{R}_\mathrm{C})\quad \frac{t \xrightarrow{w} t'}{X \xrightarrow{aw} t'}\ X \xrightarrow{a} t \in \Delta$$

$$(\mathrm{R}'_\epsilon)\quad \frac{}{X \xrightarrow{\epsilon} X} \qquad\qquad (\mathrm{R}_\mathrm{P})\quad \frac{t_1 \xrightarrow{w_1} t'_1 \quad t_2 \xrightarrow{w_2} t'_2}{t_1 \parallel t_2 \xrightarrow{w} t'_1 \parallel t'_2}\ w \in w_1 \sqcup\!\sqcup\, w_2$$

$$(\mathrm{R}_\mathrm{S})\quad \frac{t_1 \xrightarrow{w_1} t'_1}{t_1.t_2 \xrightarrow{w_1} t'_1.t_2} \qquad\qquad (\mathrm{R}'_\mathrm{S})\quad \frac{t_1 \xrightarrow{w_1} t'_1 \quad t_2 \xrightarrow{w_2} t'_2}{t_1.t_2 \xrightarrow{w_1 w_2} t'_1.t'_2}\ Const(t'_1) = \emptyset$$

Par exemple, avec les règles suivantes :

$$\Delta \stackrel{\text{def}}{=} \left\{ \begin{array}{lllllll} X & \xrightarrow{a} & (Y \parallel Z).Y & \quad & Z & \xrightarrow{c} & Z.Z \\ Y & \xrightarrow{b} & 0 & & Z & \xrightarrow{d} & 0 \end{array} \right\}$$

le terme X est à l'origine de plusieurs exécutions dont par exemple celles qui commencent par

$$X \xrightarrow{a} (Y \parallel Z).Y \xrightarrow{c} (Y \parallel (Z.Z)).Y \xrightarrow{d} (Y \parallel (0.Z)).Y \xrightarrow{c} (0 \parallel (0.Z)).Y$$

Plus généralement, la définition donnée plus haut nous permet de dériver par exemple la transition

$$X \xrightarrow{acdc} (0 \parallel (0.Z)).Y$$

Remarquons que, par rapport à des présentations habituelles (telles que [BW90]), notre définition de PA diverge sur trois points :

1. Les comportements sont engendrés par des règles de réécriture plutôt que définis par des équations récursives. C'est-à-dire qu'on écrira $\Delta = \{X \xrightarrow{a} X, X \xrightarrow{b} Y, Y \to \ldots\}$, suivant la démarche de [Mol96], plutôt que « $X \stackrel{\text{def}}{=} a.X + bY$; $Y \stackrel{\text{def}}{=} \ldots$ », à la CCS. Ces deux approches sont équivalentes à un détail près : les équations permettent la récursion non-gardée.

[2] C'est-à-dire des règles de « Structured Operational Semantics » popularisées par [Plo81].

2. On définit directement une relation $\to$ dans $E_{\mathrm{PA}} \times Act^* \times E_{\mathrm{PA}}$ plutôt que de mettre bout à bout des transitions $\xrightarrow{a}$ pour $a \in Act$. Cette approche, dite « big-steps », est un peu moins naturelle mais elle nous permettra de simplifier grandement certaines de nos preuves. Notons que nous continuerons à utiliser les abréviations $t \xrightarrow{*} t'$, $t \xrightarrow{+} t'$, et $t \to t'$ avec leur sens habituel de, respectivement, $\exists w \in Act^*, t \xrightarrow{w} t'$, $\exists w \in Act^+, t \xrightarrow{w} t'$, et $\exists a \in Act, t \xrightarrow{a} t'$.
3. Les termes ne sont pas quotientés par une congruence structurelle (c.-à-d. que nous distinguons $0.t$ et $0 \parallel t$ de t, $t_1 \parallel t_2$ de $t_2 \parallel t_1$, ceci au prix peut-être de règles de transition un peu plus complexes, et d'une écriture plus lourde pour nos termes).

Le dernier point est crucial et c'est lui qui rend notre démarche possible. (Dans la section 10.5 nous montrons comment prendre en compte de telles congruences structurelle.)

10.3 Reconnaissabilité des ensembles d'atteignabilité

Nous allons appliquer la théorie classique des automates d'arbres et des langages reconnaissables d'arbres [3] aux termes PA. En effet, les termes PA peuvent être vus comme des arbres sur la signature

$$\Sigma_{\mathrm{PA}} = \{\parallel^2, .^2, 0^0, X^0, Y^0, \ldots\}$$

(où l'exposant dénote l'arité) et un ensemble de termes $L \subseteq E_{\mathrm{PA}}$ est un langage d'arbres finis.

La théorie des langages réguliers d'arbres s'avère pertinente pour nos problèmes car le comportement des processus PA, tel que défini plus haut, est très dépendant de la structure du terme. Par exemple, on a

Lemme 10.1. [LS02] *L'ensemble* $L_{\mathrm{term}} \stackrel{def}{=} \{t \mid t \not\to\}$ *des processus* terminés *de PA forme un langage d'arbres reconnaissable, et un automate* $\mathcal{A}_{\mathrm{term}}$ *reconnaissant* L_{term} *est de taille* $O(1)$ [4].

L'atteignabilité est décidable pour PA. En fait, on dispose de résultats plus généraux résumés dans les trois théorèmes qui suivent.

Théorème 10.2. [LS02] *Si* $L \subseteq E_{\mathrm{PA}}$ *est un langage régulier d'arbres, alors* $Pred(L)$, $Pred^*(L)$ *et* $Pred^+(L)$ *sont des langages réguliers d'arbres.*

De plus, à partir d'un automate $\mathcal{A}_L$ *reconnaissant* L*, on peut construire en temps polynômial des automates reconnaissant* $Pred(L)$, $Pred^*(L)$ *et* $Pred^+(L)$ *de taille* $O(|\mathcal{A}_L|)$.

Notons que la taille des automates des prédécesseurs ne dépend pas de $|\Delta|$ et, de fait, avec un ensemble Δ infini, le théorème reste valide sauf en ce qui concerne la possibilité de construire effectivement les automates.

Théorème 10.3. [LS02] *Si* $L \subseteq E_{\mathrm{PA}}$ *est un langage régulier d'arbres, alors* $Post(L)$, $Post^*(L)$ *et* $Post^+(L)$ *sont des langages réguliers d'arbres.*

De plus, à partir d'un automate $\mathcal{A}_L$ *reconnaissant* L*, on peut construire en temps polynômial des automates reconnaissant* $Post(L)$, $Post^*(L)$ *et* $Post^+(L)$ *de taille* $O(|\mathcal{A}_L| \times |\Delta|)$.

Les Théorèmes 10.2 et 10.3 de [LS02] s'avèrent être des corollaires d'un résultat plus général :

[3] Théorie issue des travaux fondateurs de Thatcher, Wright et Doner [TW68, Don70]. On trouvera des présentations plus complètes dans des ouvrages récents [NP92, GS97, CDG+99].

[4] Pour les mesures de complexité on suppose dans toute notre présentation que Σ_{PA} est fixé et que seuls Δ, t, etc., sont des paramètres.

Théorème 10.4. [LS00d] *Les relations $\to$, $\xrightarrow{*}$ et $\xrightarrow{+}$, vues comme des parties de $E_{\mathrm{PA}} \times E_{\mathrm{PA}}$, sont des relations d'arbres reconnaissables.*

De plus, on peut construire en temps polynômial des automates $\xrightarrow{*}$, $\xrightarrow{+}$ et $\to$ de taille $O(|\Delta|)$.

Ces résultats s'étendent à des notions d'atteignabilité qui prennent en compte l'étiquette $w \in Act^*$ de la transition $t \xrightarrow{w} t'$ et, par exemple, pour chaque $w \in Act^*$, la relation $\xrightarrow{w}$ est reconnaissable.

En général, pour un langage $C \subseteq Act^*$ régulier, la relation $\xrightarrow{C}$, définie comme $\bigcup_{w\in C} \xrightarrow{w}$, n'est pas reconnaissable (en fait, on sait même que la vacuité de $Post[C](t) \stackrel{\text{def}}{=} \{t' \mid \exists w \in C, t \xrightarrow{w} t'\}$ est indécidable en général). On peut toutefois étendre le Théorème 10.4 à tous les langages C dits « décomposables » [5].

10.4 Décidabilité de la logique de transitions

10.4.1 La logique LT

LT, la *logique de transitions du premier ordre*, est la logique du premier ordre où les objets sont les processus PA et les prédicats sont les relations d'atteignabilité $\xrightarrow{*}$, $\to$ (ainsi que l'égalité et tout prédicat associé à un langage régulier, p.ex. « X apparaît dans t » (noté $X \in t$), « $\|$ apparaît dans t », « $|t|$ est impair », ...).

Par exemple, la formule suivante est une formule LT :

$$\forall u_1, u_2, u_3 \Big(\big(u_1 \xrightarrow{*} u_2 \wedge u_1 \xrightarrow{*} u_3\big) \Rightarrow \exists u_4 \big(u_2 \xrightarrow{*} u_4 \wedge u_3 \xrightarrow{*} u_4\big)\Big) \tag{10.1}$$

qui énonce la confluence de $\xrightarrow{*}$.

Il est permis d'utiliser des termes E_{PA} avec des variables libres dans les formules LT, comme par exemple dans

$$\forall u, u', v, v' \big(u.v \xrightarrow{*} u'.v' \Rightarrow u \parallel v \xrightarrow{*} u' \parallel v'\big) \tag{10.2}$$

qu'on pourrait lire « ce qui peut être fait séquentiellement peut être fait en parallèle ». L'utilisation de variables libres n'est qu'un abus de notation qui se retranscrit directement dans les prédicats de base de LT.

Un dernier exemple de formule LT est

$$\varphi(u) \stackrel{\text{def}}{=} \forall v(u \xrightarrow{*} v \wedge v \not\to) \Rightarrow \exists v'(X \notin v' \wedge u \xrightarrow{*} v' \xrightarrow{*} v). \tag{10.3}$$

Pour $t \in E_{\mathrm{PA}}$, $t \models \varphi(u)$ ssi tous les états terminés atteignables à partir de t sont atteignables via un état sans occurrence de X.

Notons que, pour une formule $\varphi(u_1, \ldots, u_n)$ et des processus $t_1, \ldots, t_n \in E_{\mathrm{PA}}$, savoir si $t_1, \ldots, t_n \models \varphi$ dépend de Δ. Pour $\varphi(u_1, \ldots, u_n)$, l'ensemble des n-uplets de termes vérifiant φ est noté $Mod_\Delta(\varphi)$, ou plus simplement $Mod(\varphi)$:

$$Mod(\varphi) \stackrel{\text{def}}{=} \{\langle t_1, \ldots, t_n\rangle \in E_{\mathrm{PA}}^{n} \mid t_1, \ldots, t_n \models \varphi(u_1, \ldots, u_n)\}.$$

Ainsi, pour une formule φ sans variable libre telle que (10.1), $Mod(\varphi) = \{\langle\rangle\}$ (où $\langle\rangle$ est le 0-uplet) si φ est valide, $Mod(\varphi)$ est vide sinon.

[5] La famille des langages décomposables contient en particulier tous les langages réguliers commutatifs et est fermée par union et concaténation [Sch99].

10.4.2 Décidabilité de LT

La décidabilité de LT est une conséquence directe du Théorème 10.4 :

Théorème 10.5. [LS00d] *Pour toute formule φ de LT, l'ensemble $Mod(\varphi)$ est une relation reconnaissable.*

Par ailleurs, à partir de Δ et de φ, on peut construire effectivement un automate $\mathcal{A}_\varphi$ reconnaissant $Mod(\varphi)$.

Preuve (Idée de la construction). La construction de $\mathcal{A}_\varphi$ se fait par induction sur la structure de φ. Les opérations booléennes se traduisent en constructions classiques pour la complémentation ou l'intersection d'automates, l'automate associé à une formule $\exists u.\psi$ s'obtient à partir de l'automate $\mathcal{A}_\psi$ par une simple projection, et les formules atomiques réutilisent l'automate du Théorème 10.4. Notons qu'en général cette construction coûte une exponentielle par alternation de quantificateurs dans φ. (De fait, le problème de décider du vide de $Mod(\varphi)$ est non-élémentaire.) □

Il est possible d'utiliser simultanément plusieurs déclarations Δ différentes, simplement en utilisant des prédicats d'atteignabilité indexés. P.ex., on énonce que tous les processus atteignables depuis X via Δ le sont aussi en utilisant Δ_1 puis Δ_2, au moyen de

$$\forall u\Big[X \xrightarrow{*}_\Delta u \;\Rightarrow\; \exists v\big(X \xrightarrow{*}_{\Delta_1} v \wedge v \xrightarrow{*}_{\Delta_2} u\big)\Big]. \tag{10.4}$$

On voit immédiatement comment adapter le Théorème 10.5 à de telles formules.

10.4.3 Spécifier avec LT

LT a été identifiée dans [CJ99], où les auteurs démontrent un équivalent du Théorème 10.5 pour les automates temporisés d'Alur et Dill.

Il est intéressant de comparer cette logique avec les logiques temporelles classiquement utilisées pour le model-checking et d'expliquer pourquoi nos résultats de décidabilité sont incomparables avec ceux de [BEH95b, BEH95a].

LT permet directement d'exprimer le fragment $B(\mathsf{EF}, \mathsf{EX})$ de CTL. Ce fragment, appelé aussi EF, est le plus grand fragment identifié pour lequel le model-checking de PA est décidable [May01].

LT a la possibilité de faire directement référence à l'*identité* des états, ce que ne peut pas faire CTL (ni CTL^* ou la bisimulation). Ainsi des formules telles que (10.1) n'ont pas d'équivalent en CTL^*.

Toutefois LT ne permet pas d'exprimer toutes les formules temporelles : la notion de chemin n'est pas présente dans la logique de transitions. Ainsi, il n'est pas possible de quantifier sur les états visités le long d'un chemin (comme dans $\mathsf{E}\varphi\mathsf{U}\psi$ ou dans $\mathsf{EG}\varphi$).

10.4.4 Model-checking, vérification paramétrée et mesure de modèles

Calculer $Mod(\varphi)$ est plus général que simplement décider si φ est valide, ou que dire si $t \models \varphi(u)$ pour t donné (ce qui est la forme habituelle des problèmes de model-checking).

Dans les applications de vérification, la possibilité de calculer $Mod(\varphi)$ et de le représenter sous une forme symbolique adéquate (un automate d'arbre dans notre cas) permet de mettre en œuvre une approche qui généralise élégamment ce qu'on appelle la *vérification paramétrée* [6] et la *mesure de modèles* [7].

Nous allons donner un exemple, qui est un peu artificiel dans la mesure où notre présentation de PA et de la logique LT ne met pas en œuvre la notion de coûts des transitions qui est introduite dans [LS00d]. Notons X^n pour le terme

$$\overbrace{X \parallel (X \parallel (X \cdots \parallel X) \ldots)}^{n \text{ copies de } X}.$$

Les ensembles $L \stackrel{\text{def}}{=} \{X^n \mid n = 0, 1, 2, \ldots\}$ et $L' \stackrel{\text{def}}{=} \{Y^n \mid n = 0, 1, 2, \ldots\}$ sont des langages réguliers d'arbres.

Soit alors une formule $\varphi(u, v)$ de LT qui compare le comportement de u et v. P.ex.

$$\varphi(u, v) \stackrel{\text{def}}{=} \forall z\big((v \xrightarrow{*} z \wedge z \xrightarrow{+} z) \Rightarrow u \xrightarrow{*} z\big)$$

qui énonce que « les boucles de v sont des boucles de u ».

En calculant $Mod(u \in L \wedge v \in L' \wedge \varphi(u, v))$ on obtient les valeurs de $\langle n, m\rangle$ pour lesquelles $\varphi(X^n, Y^m)$ est vraie (c'est d'ailleurs un semilinéaire). Par la même méthode, on peut calculer quelles valeurs de m rendent $\varphi(u, Y^m)$ satisfaisable, ou valide, pour les X^n.

10.5 Congruence structurelle pour PA

La congruence canonique sur les termes PA est l'équivalence $\equiv$ induite par les équations suivantes :

$$\begin{array}{lrcl}
(C_\parallel) & t \parallel t' & \equiv & t' \parallel t \\
(A_\parallel) & (t \parallel t') \parallel t'' & \equiv & t \parallel (t' \parallel t'') \\
(A_.) & (t.t').t'' & \equiv & t.(t'.t'')
\end{array}
\qquad
\begin{array}{lrcl}
(N_1) & t.0 & \equiv & t \\
(N_2) & 0.t & \equiv & t \\
(N_3) & t \parallel 0 & \equiv & t \\
(N_4) & 0 \parallel t & \equiv & t
\end{array}$$

La plupart des travaux sur PA ne considèrent que des termes modulo $\equiv$. Ce faisant, ils bénéficient d'une notation allégée pour les termes, et font d'ores et déjà abstraction de certains détails structurels qui ne sont pas pertinents : ils se rapprochent alors de la sémantique des termes.

Notre approche à base d'automates d'arbres ne peut pas traiter directement ces termes quotientés. Mais nous allons voir comment il est néanmoins possible de répondre à des questions mettant en jeu l'équivalence $\equiv$ (ou une autre). Pour cela, nous allons comparer notre démarche à celle de [May97d, May97b].

[6] Dans la vérification paramétrée, certaines parties du système examiné ne sont pas fixées et apparaissent sous la forme de paramètres. On vérifie ensuite que le système est correct pour toutes les valeurs possibles des paramètres (ou on calcule pour quelles valeurs le système est correct).

[7] Ou « model measuring » en anglais. Dans la mesure de modèles, c'est la propriété à vérifier qui contient des paramètres. La procédure de vérification du modèle calcule alors pour quelles valeurs des paramètres la propriété est satisfaite [CY92, CG96, AELP99, ET99].

Notons $[t]_\equiv$ l'ensemble $\{t' \mid t \equiv t'\}$ et $t \stackrel{a}{\Rightarrow} t'$ quand il existe u, u' tels que $t \equiv u \xrightarrow{a} u' \equiv t'$ ($\stackrel{a}{\Rightarrow}$ est donc la composition $\equiv \circ \xrightarrow{a} \circ \equiv$). La relation de transition utilisée par Mayr coïncide avec une relation définie via

$$[t]_\equiv \xrightarrow{a} [u]_\equiv \quad \stackrel{\text{def}}{\Leftrightarrow} \quad t \stackrel{a}{\Rightarrow} u \quad \Leftrightarrow \quad \exists t' \in [t]_\equiv, u' \in [u]_\equiv \text{ tq } t' \xrightarrow{a} u'. \tag{10.5}$$

Nous écrirons $\mathrm{PA}_\equiv$ pour désigner le système de transitions où les états sont les classes $[t]_\equiv$ et où les transitions sont données par (10.5). Ce système est une image homomorphe de $\langle E_{\mathrm{PA}}, \Rightarrow\rangle$.

Le problème d'atteignabilité résolu par Mayr est celui du système de transitions $\mathrm{PA}_\equiv$, et nous allons montrer comment le résoudre dans notre cadre. Notre approche est plus générale que celle de Mayr au sens où nous pouvons définir $\mathrm{PA}_\equiv$ à partir de PA, alors que si on raisonne modulo $\equiv$ dès le départ, l'information permettant de revenir à PA est irrémédiablement perdue.

10.5.1 Congruence structurelle et régularité

$(A_.)$, $(C_\|)$ et $(A_\|)$ sont les lois d'associativité-commutativité satisfaites par . et $\|$. On parlera d'*axiomes permutatifs* et on écrira $t \leftrightarrows u$ quand t et u sont permutativement équivalents.

(N_1) à (N_4) sont les lois de l'élément neutre. On parlera d'*axiomes de simplification* et on écrira $t \searrow u$ quand u s'obtient par simplification de t, c.-à-d. u s'obtient à partir de t en appliquant (un nombre quelconque de fois) les axiomes de simplification *de gauche à droite*. Notons que $t \leftrightarrows t_1$ et $t \searrow t_2$ impliquent $|t_1| = |t| \geq |t_2|$ et que $\searrow$ est un ordre bien fondé. $\swarrow$ dénote $(\searrow)^{-1}$ et la *forme normale simplifiée* de t, notée $t\!\downarrow$, est l'unique u obtenu en simplifiant t autant que possible.

On note $t \stackrel{1}{\leftrightarrows} u$ (resp. $t \searrow^1 u$) quand on passe de t à u en appliquant *une fois* un axiome permutatif (resp. un axiome de simplification), de sorte que $\leftrightarrows\, = (\stackrel{1}{\leftrightarrows})^*$ et $\searrow\, = (\searrow^1)^*$. De plus, ${}^1\!\swarrow$ désigne $(\searrow^1)^{-1}$.

Proposition 10.6. [LS02] *Les trois propositions suivantes sont équivalentes :*

1. $t_1 \equiv t_2$ *;*
2. *il existe* u_1, u_2 *tels que* $t_1 \searrow u_1 \leftrightarrows u_2 \swarrow t_2$ *;*
3. $t_1\!\downarrow\, \leftrightarrows t_2\!\downarrow$.

Démonstration. On a (3) $\Rightarrow$ (2) $\Rightarrow$ (1) par définition. L'implication « (1) $\Rightarrow$ (2) » s'obtient par des techniques classiques de commutation de relations de réduction. Plus précisément, les deux commutations suivantes :

– si $t\, {}^1\!\swarrow \stackrel{1}{\leftrightarrows} v$ alors $t \leftrightarrows {}^1\!\swarrow v$;
– si $t\, {}^1\!\swarrow \searrow^1 v$ alors $t \searrow \swarrow v$;

ainsi que les commutations symétriques permettent de transformer toute séquence $t_1 \rho_1 \rho_2 \ldots \rho_n t_2$ où $\rho_i \in \{\stackrel{1}{\leftrightarrows}, \searrow^1, {}^1\!\swarrow\}$, en une séquence $t_1 \searrow \leftrightarrows \swarrow t_2$.

L'implication « (1) $\Rightarrow$ (3) » s'obtient en appliquant les mêmes commutations a une séquence de la forme $t_1\!\downarrow \swarrow t_1 \equiv t_2 \searrow t_2\!\downarrow$. On obtient $t_1\!\downarrow \searrow u_1 \leftrightarrows u_2 \swarrow t_2\!\downarrow$ où forcément $u_i = t_i\!\downarrow$ par définition des formes normales simplifiées. $\square$

Ce résultat nous permet de réduire des problèmes mettant en jeu l'équivalence $\equiv$ en problèmes sur $\leftrightarrows$ et $\searrow$. Notons que

Lemme 10.7. *Pour tout t, l'ensemble $[t]_{\leftrightarrows} \stackrel{def}{=} \{u \mid t \leftrightarrows u\}$ est un langage régulier.*

Par ailleurs, on peut construire un automate reconnaissant $[t]_{\leftrightarrows}$ de taille $O(m.(m/2)!)$ si $|t| = m$.

Démonstration. $[t]_{\leftrightarrows}$ est fini et n'a pas plus de $(m/2)!$ éléments. $\square$

Remarquons que si $L \subseteq E_{\mathrm{PA}}$ est régulier, $[L]_{\leftrightarrows} \stackrel{\mathrm{def}}{=} \bigcup_{t \in L} [t]_{\leftrightarrows}$ n'est pas forcément régulier [GD89].

Les axiomes de simplification se comportent mieux : notons $[L]_{\searrow}$ pour $\{u \mid \exists t \in L, t \searrow u\}$, $[L]_{\swarrow}$ pour $\{u \mid \exists t \in L, u \searrow t\}$, et $[L]{\downarrow}$ pour $\{t{\downarrow} \mid t \in L\}$. On a alors :

Lemme 10.8. [LS02] *Si L est un langage d'arbres régulier, alors les ensembles $[L]_{\swarrow}$, $[L]_{\searrow}$, et $[L]{\downarrow}$ sont réguliers.*

Par ailleurs, à partir d'un automate $\mathcal{A}_L$ reconnaissant L, on peut construire en temps polynômial des automates de taille $O(|\mathsf{A}|)$ pour ces trois langages.

10.5.2 Congruence structurelle et comportement

Si les travaux classiques travaillent sur PA modulo congruence $\equiv$ c'est parce que cette congruence est compatible avec les comportements des processus PA, au sens où $t \equiv u$ implique que t et u sont bisimilaires (noté $t \sim u$) [BW90].

Il se trouve qu'on a une propriété encore plus forte :

Proposition 10.9. [LS02] *$\equiv$ est une relation de bisimulation, c.-à-d. que pour tous $t \equiv t'$ et $t \xrightarrow{a} u$ il existe un $u' \equiv u$ tel que $t' \xrightarrow{a} u'$.*

On en déduit

Lemme 10.10. [LS02] *Pour tout $w \in Act^*$, $t \stackrel{w}{\Rightarrow} u$ ssi $t \xrightarrow{w} u'$ pour un $u' \equiv u$.*

Démonstration. Par induction sur la longueur de w, et en utilisant la Proposition 10.9. $\square$

10.5.3 Atteignabilité modulo $\equiv$

Il est maintenant facile de prouver la décidabilité du problème de l'atteignabilité dans $\mathrm{PA}_{\equiv}$. En effet, $t \stackrel{*}{\Rightarrow} u$ ssi $Post^*(t) \cap [u]_{\equiv} \neq \emptyset$. Or $[u]_{\equiv}$ et $Post^*(t)$ sont des langages réguliers qu'on peut construire effectivement. On peut donc décider s'ils ont une intersection non-vide. Cet algorithme simple demande un temps exponentiel à cause de la taille de $[u]_{\equiv}$.

On peut améliorer ce résultat par

Théorème 10.11. [LS02] *Le problème d'atteignabilité dans $\mathrm{PA}_{\equiv}$ est NP-complet.*

Ce résultat est annoncé dans [May97d] mais, pour la borne supérieure, la preuve complète n'est pas donnée (le sens « NP-difficile » est prouvé dans [Esp97b] pour BPP). Or il est facile de donner un algorithme non-déterministe en temps polynômial dans notre cadre.

Démonstration (de l'appartenance à NP). $t \stackrel{*}{\Rightarrow} u$ ssi $t \xrightarrow{*} t'$ pour un t' tel que $t'{\downarrow} \leftrightarrows u{\downarrow}$. Notons t'' pour $t'{\downarrow}$ et remarquons que $|t''| \leq |u|$.

Pour décider si $t \stackrel{*}{\Rightarrow} u$, il suffit de de calculer $u{\downarrow}$, de deviner de façon non-déterministe une permutation t'' de $u{\downarrow}$, de construire des automates $\mathcal{A}_1$ reconnaissant $[t'']_{\searrow}$ et $\mathcal{A}_2$ reconnaissant $Post^*(t)$. Ces automates sont de taille polynomiale. Il reste à vérifier que $\mathcal{A}_1$ et $\mathcal{A}_2$ ont une intersection non-vide pour savoir si le t' cherché existe. $\square$

Il est possible de résoudre d'autres problèmes que l'atteignabilité :

Théorème 10.12. [LS02] *La finitude de l'ensemble d'atteignabilité est décidable en temps polynômial pour* $\mathrm{PA}_{\equiv}$.

Démonstration. $[t]_{\equiv}$ peut atteindre un nombre infini de termes différents dans $\mathrm{PA}_{\equiv}$ ssi t peut atteindre dans PA un nombre infini de termes deux à deux non équivalents (au sens de $\equiv$). Mais puisque les axiomes permutatifs ne permettent qu'un nombre fini de variantes pour un terme donné, on en déduit que $Post^*(L)$ contient une infinité de termes deux à deux non-équivalents ssi $[Post^*(L)]{\downarrow}$ est fini. □

Pour le model-checking des formules EF, il est encore plus simple de passer de $\mathrm{PA}_{\equiv}$ à PA. En effet, si on suppose que les propositions atomiques sont interprétées de façon équivalente sur PA et $\mathrm{PA}_{\equiv}$, alors on a $t \models \varphi$ dans $\langle E_{\mathrm{PA}}, \rightarrow \rangle$ ssi $t \models \varphi$ dans $\langle E_{\mathrm{PA}}, \Rightarrow \rangle$ pour toute formule φ compatible avec la bisimulation, et en particulier pour les formules de CTL^*.

10.6 Notes bibliographiques

Les algèbres de processus (CCS et ses dérivés) sont des modèles pour la sémantique des processus concurrents, et forment un domaine à part depuis leur invention par Milner [Mil80, Mil83, Mil89] et d'autres [Hoa78, AB84, BK84, Hoa85, BK85, BW90].

Ces algèbres ont la puissance des machines de Turing [Tau89, Vaa93] et les questions de décidabilité n'étaient étudiées que dans le cas des systèmes finis [KS90, CPS93], ceci jusqu'à ce que [BBK93] montre que la bisimulation est décidable pour les processus normés de BPA [8]. Il s'agissait d'un résultat surprenant car l'équivalence des traces est indécidable pour BPA (en fait la bisimulation est la seule équivalence décidable pour BPA [GH94]). De nombreux travaux ont ensuite montré comment décider la bisimulation sur BPA et BPP [9] sans l'hypothèse de norme [CHM94, Mol96, BE97, JM99].

Ces travaux se sont prolongés par l'étude de la décidabilité d'autres questions (p.ex. le model-checking de LTL [EHRS00], de CTL [EK95], la finitude [Kuč99], etc.), l'évaluation plus précise de la complexité intrinsèque des problèmes (p.ex. [May00a]) et l'exploration d'algèbres plus riches (PA, PRS [May00b], etc.)

PA combine les constructions de BPP et de BPA. Pendant plusieurs années les chercheurs qui travaillaient avec succès sur BPP et BPA ne parvenaient pas à étendre leurs méthodes à PA. Ceci jusqu'à ce que R. Mayr développe une méthode « de tableaux » pour les problèmes d'atteignabilité dans PA [May97d, May97b, May97a]. Ce n'est qu'avec [HJ99] que la bisimulation a été partiellement montrée décidable pour PA.

Nos résultats sur PA ont été inspirés par [FWW97, BEM97] qui redémontraient que l'ensemble d'atteignabilité d'un pushdown system est reconnaissable. En fait, pour ces systèmes la relation $\xrightarrow{*}$ est une transduction rationnelle [Cau92]. De nombreux résultats de cette forme existent : la relation $\xrightarrow{*}$ est semilinéaire pour les BPP [FO97, Esp97b], définissable dans la théorie additive des réels [10] pour les automates temporisés [CJ99], reconnaissable pour les GRS [11] [DT90] et les systèmes de réécriture préfixe de mots [Cau00].

[8] *BPA*, « Basic Parallel Algebra », est l'algèbre combinant non-déterminisme, composition séquentielle et récursivité. Les processus *normés* sont ceux qui peuvent terminer.

[9] *BPP*, « Basic Parallel Processes », est l'algèbre combinant non-déterminisme, composition parallèle (sans synchronisation) et récursivité. BPP correspond aux réseaux de Petri sans synchronisation [Esp97b].

[10] Une variante de la logique de Presburger pour les réels.

[11] Les « ground rewrite systems », c.-à-d. les systèmes de réécriture de termes où les parties gauches sont sans variable.

La reconnaissabilité de $\xrightarrow{*}$ conduit à une procédure de décision pour la logique de transitions. C'est [CJ99] qui a le premier présenté cette approche et comparé le pouvoir d'expressivité des logiques de transitions avec celui des logiques temporelles classiques.

Conclusion & Perspectives

La vérification est une technologie aux applications pratiques importantes, touchant à des enjeux industriels très larges. C'est aussi un sujet de recherche très actif, qui a donné lieu à une théorie très riche et originale, qui reste toujours très vivante aujourd'hui, et dont ce mémoire a présenté un survol, évidemment très partiel puisque essentiellement axé sur mes travaux.

On peut comprendre d'où viennent la richesse et l'originalité du sujet en constatant qu'il se trouve à une intersection triple :

- les notions théoriques sont principalement issues de la *logique* et de la *théorie des automates*, deux domaines riches et fondamentaux qui se sont rapprochés ces dernières années ;
- les applications pratiques très importantes donnent tout son poids à la dimension *algorithmique* du problème, allant de la théorie de la complexité à la réalisation de model-checkers efficaces, capables de traiter des systèmes de plus en plus complexes ;
- enfin la nécessité de modéliser le système à vérifier fait que la théorie de la vérification relève aussi de la pratique de la *programmation*, touchant aussi bien à la sémantique des langages de programmation, qu'au génie logiciel et aux pratiques de programmation (comme le fait, de façon plus générale, toute la problématique des méthodes formelles en programmation).

Il n'y a guère de doute que la théorie de la vérification va rester un domaine riche et vivant dans les prochaines années. On peut d'ores et déjà relever deux tendances de fond :

Model-checking et théorie des contraintes : On constate que les techniques et les résultats issus de la théorie des contraintes jouent un rôle de plus en plus important dans le model-checking. Ainsi de nombreux résultats de décidabilité pour les systèmes infinis reposent sur des représentations des données (p.ex. les automates d'arbres) ou des fragments logiques (p.ex. l'arithmétique de Presburger) déjà largement explorés pour la résolution de contraintes.

Par ailleurs, les techniques de représentation utilisées dans le model-checking symbolique restent encore aujourd'hui assez limitées (voire naïves pour certaines utilisations des BDDs) et sont amenées à s'enrichir en réutilisant les idées développées dans la résolution de contraintes symboliques.

Vérification de modèles étendus : Les techniques de model-checking ont d'abord été mises au point et implémentées pour vérifier des structures de Kripke finies (représentées de façon énumérative, puis symbolique). Ces techniques ont été étendues (en théorie) de façon à prendre en compte des systèmes étendus. Dans cette direction, les principaux succès pratiques restent le passage aux systèmes temporisés [HNSY94] puis hybrides [ACH$^+$95].

Dans ces deux exemples, le succès est d'abord dû à l'identification d'applications réelles pour lesquelles les limites imposées par la complexité des algorithmes de vérification (quelques horloges seulement, et un ensemble d'états de contrôle à la portée des méthodes énumératives) n'est pas une limitation.

D'autres extensions du model-checking classique ont été étudiées récemment : les processus mobiles, les systèmes probabilistes, les systèmes temporisés répartis, ainsi que certaines classes de réseaux de Petri non bornés, et sans doute d'autres extensions verront le jour.

Bibliographie

[AB84] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30(1) :91–131, 1984.

[ABS90] C. Autant, Z. Belmesk, and Ph. Schnoebelen. A net-theoretic approach to the efficient implementation of the FP2 parallel language. In *Proc. 11th Int. Conf. Application and Theory of Petri Nets (ICATPN'90), Paris, France, June 1990*, pages 451–470, 1990.

[ABS91] C. Autant, Z. Belmesk, and Ph. Schnoebelen. Strong bisimilarity on nets revisited. In *Proc. Parallel Architectures and Languages Europe (PARLE'91), Eindhoven, NL, June 1991, vol. II : Parallel Languages*, volume 506 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 1991.

[AC88] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2) :57–66, 1988.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, Pei-Hsin Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1) :3–34, 1995.

[AČJT00] P. A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1/2) :109–127, 2000.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[AELP99] R. Alur, K. Etessami, S. La Torre, and D. Peled. Parametric temporal logic for "model measuring". In *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99), Prague, Czech Republic, July 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 1999.

[AH92] R. Alur and T. A. Henzinger. Logics and models of real time : A survey. In *Real-Time : Theory in Practice, Proc. REX Workshop, Mook, NL, June 1991*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.

[AH94] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1) :181–203, 1994.

[AI01] M. Adler and N. Immerman. An $n!$ lower bound on formula size. In *Proc. 16th IEEE Symp. Logic in Computer Science (LICS'2001), Boston, MA, USA, June 2001*, pages 197–206. IEEE Comp. Soc. Press, 2001.

[AJ96a] P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1) :71–90, 1996.

[AJ96b] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2) :91–101, 1996.

[AK95] P. A. Abdulla and M. Kindahl. Decidability of simulation and bisimulation between lossy channel systems and finite state systems. In *Proc. 6th Int. Conf. Theory of Concurrency (CONCUR'95), Philadelphia, PA, USA, Aug. 1995*, volume 962 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 1995.

[AL99] L. Aceto and F. Laroussinie. Is your model checker on time ? In *Proc. 24th Int. Symp. Math. Found. Comp. Sci. (MFCS'99), Szklarska Poreba, Poland, Sep. 1999*, volume 1672 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1999.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford Univ., August 1991. Available as Tech. Report STAN-CS-91-1378.

[AN82] A. Arnold and M. Nivat. Comportements de processus. In *Actes du Colloque AFCET «Les Mathématiques de l'Informatique»*, pages 35–68, 1982.

[AN00] P. A. Abdulla and A. Nylén. Better is better than well : On efficient verification of infinite-state systems. In *Proc. 15th IEEE Symp. Logic in Computer Science (LICS'2000), Santa Barbara, CA, USA, June 2000*, pages 132–140. IEEE Comp. Soc. Press, 2000.

[AN01] P. A. Abdulla and A. Nylén. Timed Petri nets and BQOs. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'2001), Newcastle upon Tyne, UK, June 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2001.

[APS94] C. Autant, W. Pfister, and Ph. Schnoebelen. Place bisimulations for the reduction of labeled Petri nets with silent moves. In *Proc. 6th Int. Conf. on Computing and Information (ICCI'94), Trent University, Canada, May 1994*, pages 230–246, 1994.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Collection Études et Recherches en Informatique. Masson, Paris, 1992.

[AS92] C. Autant and Ph. Schnoebelen. Place bisimulations in Petri nets. In *Proc. 13th Int. Conf. Application and Theory of Petri Nets (ICATPN'92), Sheffield, UK, June 1992*, volume 616 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 1992.

[Aut93] C. Autant. *Réseaux de Petri pour la sémantique et l'implémentation de processus parallèles.* Thèse de Doctorat, I.N.P. de Grenoble, France, May 1993.

[BBK93] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3) :653–682, 1993.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2) :115–131, 1988.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, 98(2) :142–170, 1992.

[BE97] O. Burkart and J. Esparza. More infinite results. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, volume 5 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 1997.

[BEH95a] A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *Proc. 10th IEEE Symp. Logic in Computer Science (LICS'95), San Diego, CA, USA, June 1995*, pages 123–133. IEEE Comp. Soc. Press, 1995.

[BEH95b] A. Bouajjani, R. Echahed, and P. Habermehl. Verifying infinite state processes with sequential and parallel composition. In *Proc. 22nd ACM Symp. Principles of Programming Languages (POPL'95), San Francisco, CA, USA, Jan. 1995*, pages 95–106, 1995.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata : Application to model-checking. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3) :109–137, 1984.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1) :77–121, 1985.

[BK89] J. A. Bergstra and J. W. Klop. Process theory based on bisimulation semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Proc. REX School/Workshop, Noordwijkerhout, NL, May-June 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer, 1989.

[BK98] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods in System Design*, 13(3) :253–288, 1998.

[BLS00] B. Bérard, A. Labroue, and Ph. Schnoebelen. Verifying performance equivalence for Timed Basic Parallel Processes. In *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 35–47. Springer, 2000.

[BM99] A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In *Proc. 16th Ann. Symp. Theoretical Aspects of Computer Science (STACS'99), Trier, Germany, Mar. 1999*, volume 1563 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 1999.

[BM01] J. C. M. Baeten and C. A. Middelburg. Process algebra with timing : real time and discrete time. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 10, pages 627–684. Elsevier Science, 2001.

[BPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20 :207–226, 1983.

[BRDV90] G. Boudol, V. Roy, R. De Simone, and D. Vergamini. Process calculi, from theory to practice : Verification tools. In *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89), Grenoble, June 1989*, volume 407 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1990.

[BS90] Z. Belmesk and Ph. Schnoebelen. Une généralisation des présentations FP2 basée sur la théorie des réseaux. Research Report 832-I, LIFIA-IMAG, Grenoble, France, November 1990.

[BS94] O. Burkart and B. Steffen. Pusdown processes : parallel composition and model checking. In *Proc. 5th Int. Conf. Theory of Concurrency (CONCUR'94), Uppsala, Sweden, Aug. 1994*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 1994.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 1990.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1) :61–86, 1992.

[Cau00] D. Caucal. On word rewriting systems having a rational derivation. In *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 48–62, 2000.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89), Grenoble, June 1989*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1990.

[CC00] H. Comon and V. Cortier. Flatness is not a weakness. In *Proc. 14th Int. Workshop Computer Science Logic (CSL'2000), Fischbachau, Germany, Aug. 2000*, volume 1862 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2000.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. IEEE Int. Conf.*

Systems, Man and Cybernetics (SMC'2000), Nashville, TN, USA, Oct. 2000, pages 2449–2454, 2000.

[CDG+99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 1997–99. A preliminary version of this electronic book is available at `http://www.grappa.univ-lille3.fr/tata`.

[CDP+00] G. Canet, B. Denis, A. Petit, O. Rossi, and Ph. Schnoebelen. Un cadre pour la vérification automatique de programmes IL. In *Actes 1ère Conf. Int. Francophone d'Automatique (CIFA'2000), Lille, France, July 2000*, pages 693–698. UCIS, Villeneuve d'Ascq, France, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Logics of Programs Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CFP96] G. Cécé, A. Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1) :20–31, 1996.

[CG87] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. In *Ann. Rev. Comput. Sci.*, volume 2, pages 269–290. Annual Reviews Inc., 1987.

[CG96] S. Campos and O. Grumberg. Selective quantitative analysis and interval model checking : Verifying different facets of a system. In *Proc. 8th Int. Conf. Computer Aided Verification (CAV'96), New Brunswick, NJ, USA, July-Aug. 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 1996.

[CGH97] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1) :47–71, 1997.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, 1994.

[CHM94] S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4) :233–242, 1994.

[CJ99] H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, Aug. 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 1999.

[CLS91] H. Comon, D. Lugiez, and Ph. Schnoebelen. A rewrite-based type discipline for a subset of computer algebra. *Journal of Symbolic Computation*, 11(4) :349–368, 1991.

[CMP92] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Proc. 19th Int. Coll. Automata, Languages, and Programming (ICALP'92), Vienna, Austria, July 1992*, volume 623 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 1992.

[Cor00] F. Corradini. Absolute versus relative time in process algebras. *Information and Computation*, 156(1/2) :122–172, 2000.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench : A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1) :36–72, 1993.

[CS91] F. Cherief and Ph. Schnoebelen. τ-bisimulations and full abstraction for refinement of actions. *Information Processing Letters*, 40(4) :219–222, 1991.

[CY92] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proc. 3rd Int. Workshop Computer Aided Verification (CAV'91), Aalborg, Denmark, July 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 399–409. Springer, 1992.

[Dam99] D. R. Dams. Flat fragments of CTL and CTL* : Separating the expressive and distinguishing powers. *Logic Journal of the IGPL*, 7(1) :55–78, 1999.

[DCR$^+$00] O. De Smet, S. Couffin, O. Rossi, G. Canet, J.-J. Lesage, Ph. Schnoebelen, and H. Papini. Safe programming of PLC using formal verification methods. In *Proc. 4th Int. PLCopen Conf. on Industrial Control Programming (ICP'2000), Utrecht, The Netherlands, Oct. 2000*, pages 73–78. PLCOpen, Zaltbommel, The Netherlands, 2000.

[DFS98] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.

[DG99] V. Diekert and P. Gastin. An expressively complete temporal logic without past tense operators for Mazurkiewicz traces. In *Proc. 13th Int. Workshop Computer Science Logic (CSL'99), Madrid, Spain, Sep. 1999*, volume 1683 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 1999.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2) :111–149, 1997.

[DH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2) :83–133, 1984.

[Dic13] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with r distinct prime factors. *Amer. Journal Math.*, 35 :413–422, 1913.

[DJS99] C. Dufourd, P. Jančar, and Ph. Schnoebelen. Boundedness of Reset P/T nets. In *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99), Prague, Czech Republic, July 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 301–310. Springer, 1999.

[DMV90] R. De Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. In *Proc. Theories of Concurrency (CONCUR'90), Amsterdam, NL, Aug. 1990*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990.

[Don70] J. E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4(5) :406–451, 1970.

[DR00] G. Delzanno and J.-F. Raskin. Symbolic representation of upward-closed sets. In *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2000.

[DS02] S. Demri and Ph. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Information and Computation*, 2002. To appear.

[DSC99] Xiaoqun Du, S. Smolka, and R. Cleaveland. Local model checking and protocol analysis. *Journal of Software Tools for Technology Transfer*, 2(3) :219–241, 1999.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symp. Logic in Computer Science (LICS'90), Philadelphia, PA, USA, June 1990*, pages 242–248. IEEE Comp. Soc. Press, 1990.

[Duf98] C. Dufourd. Réseaux de Petri avec reset/transfert : Décidabilité et indécidabilité. Thèse de Docteur en Sciences de l'École Normale Supérieure de Cachan, October 1998.

[DV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2) :458–487, 1995.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Coll. Automata, Languages and Programming (ICALP'80), Noordwijkerhout, NL, Jul. 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 352–359. IEEE Comp. Soc. Press, 1999.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited : On branching versus linear time. In *Proc. 10th ACM Symp. Principles of Programming Languages (POPL'83), Austin, TX, USA, Jan. 1983*, pages 127–140, 1983.

[EH85] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1) :1–24, 1985.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited : On branching versus linear time temporal logic. *Journal of the ACM*, 33(1) :151–178, 1986.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[EK95] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and basic parallel processes. In *Proc. 7th Int. Conf. Computer Aided Verification (CAV'95), Liège, Belgium, July 1995*, volume 939 of *Lecture Notes in Computer Science*, pages 353–366. Springer, 1995.

[EK99] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proc. 2nd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'99), Amsterdam, The Netherlands, Mar. 1999*, volume 1578 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 1999.

[EL87] E. A. Emerson and Chin-Laung Lei. Modalities for model checking : Branching time logic strikes back. *Science of Computer Programming*, 8(3) :275–306, 1987.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[EMSS92] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4) :331–352, 1992.

[EP00] J. Esparza and A. Podelski. Efficient algorithms for $pre*$ and $post*$ on interprocedural parallel flow graphs. In *Proc. 27th ACM Symp. Principles of Programming Languages (POPL'2000), Boston, MA, USA, Jan. 2000*, pages 1–11, 2000.

[Ers58] A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8) :3–6, 1958.

[Esp97a] J. Esparza. Decidability of model checking for infinite-sate concurrent systems. *Acta Informatica*, 34(2) :85–107, 1997.

[Esp97b] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31(1) :13–25, 1997.

[ET97] E. A. Emerson and R. J. Trefler. Generalized quantitative temporal reasoning : An automata-theoretic approach. In *Proc. 7th Int. Joint Conf. Theory and Practice of Software Development (TAPSOFT'97), Lille, France, Apr. 1997*, volume 1214 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 1997.

[ET99] E. A. Emerson and R. J. Trefler. Parametric quantitative temporal reasoning. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 336–343. IEEE Comp. Soc. Press, 1999.

[Fer89] J.-Cl. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3) :219–236, 1989.

[Fin85] A. Finkel. Une généralisation des théorèmes de Higman et de Simon aux mots infinis. *Theoretical Computer Science*, 38(1) :137–142, 1985.

[Fin87] A. Finkel. A generalization of the procedure of Karp and Miller to well structured transition systems. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP'87), Karlsruhe, FRG, July 1987*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 1987.

[Fin90] A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2) :144–179, 1990.

[Fin93] A. Finkel. The minimal coverability graph for Petri nets. In *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer, 1993.

[Fin94] A. Finkel. Decidability of the termination problem for completely specificied protocols. *Distributed Computing*, 7(3) :129–135, 1994.

[FO97] L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4) :305–335, 1997.

[FPS00] A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems. In *Proc. 11th Int. Conf. Concurrency Theory (CONCUR'2000), University Park, PA, USA, Aug. 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2000.

[FS98] A. Finkel and Ph. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proc. 3rd Latin American Theoretical Informatics Symposium (LATIN'98), Campinas, Brazil, Apr. 1998*, volume 1380 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 1998. Journal version as [FS01].

[FS01] A. Finkel and Ph. Schnoebelen. Well structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2) :63–92, 2001.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97), Bologna, Italy, July 1997*, volume 9 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 1997.

[Gab89] D. Gabbay. The declarative past and imperative future : Executable temporal logic for interactive systems. In *Proc. Workshop Temporal Logic in Specification, Altrincham, UK, Apr. 1987*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1989.

[Gar89] H. Garavel. Compilation of LOTOS abstract data types. In *Proc. 2nd Int. Conf. Formal Description Techniques (FORTE'89), Vancouver, Canada, Dec. 1989*, pages 147–162. North-Holland, 1989.

[GD89] R. Gilleron and A. Deruyver. The reachability problem for ground TRS and some extensions. In *Proc. Int. Joint Conf. Theory and Practice of Software Development (TAPSOFT'89), Barcelona, Spain, Mar. 1989, Vol. 1*, volume 351 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 1989.

[GH94] J. F. Groote and H. Hüttel. Undecidable equivalences for Basic Process Algebra. *Information and Computation*, 115(2) :354–371, 1994.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation : P-Completeness Theory.* Oxford Univ. Press, 1995.

[GKP92] U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In *Proc. 3rd Int. Conf. Concurrency Theory (CONCUR'92), Stony Brook, NY, USA, Aug. 1992*, volume 630 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 1992.

[Gla01] R. J. van Glabbeek. The linear time – branching time spectrum I. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier Science, 2001.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80), Las Vegas, NV, USA, Jan. 1980*, pages 163–173, 1980.

[GRS95] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *Theoretical Computer Science*, 140(1) :73–94, 1995.

[GS85] Y. Gurevich and S. Shelah. The decision problem for branching time logic. *The Journal of Symbolic Logic*, 50(3) :668–681, 1985.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, 1997.

[Gur85] Y. Gurevich. Monadic second-order theories. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, chapter XIII, pages 479–506. Springer, 1985.

[GW89] R. J. van Glabbeek and W. P. Weijland. Refinement in branching time semantics. In *Proc. Int. Conf. Algebraic Methodology and Software Technology (AMAST'89), Iowa City, Iowa, USA, May 1989*, pages 197–201, 1989. Available as CWI Report CS-R8922.

[GW96] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3) :555–600, 1996.

[Hac76] M. Hack. Decidability questions for Petri nets. PhD Thesis MIT/LCS/TR-161, Massachusetts Institute of Technology, Lab. for Computer Science, June 1976.

[Hal95] J. Y. Halpern. The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic. *Artificial Intelligence*, 75(2) :361–372, 1995.

[HC96] G. E. Hughes and M. J. Cresswell. *A new introduction to modal logic*. Routledge, London, 1996.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7) :326–336, 1952.

[HJ99] Y. Hirshfeld and M. Jerrum. Bisimulation equivalence is decidable for normed process algebra. In *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99), Prague, Czech Republic, July 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 412–421. Springer, 1999.

[HKV97] D. Harel, O. Kupferman, and M. Y. Vardi. On the complexity of verifying concurrent transition systems. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 1997.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1) :137–161, 1985.

[HM00] T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, France, Feb. 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 13–34. Springer, 2000.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2) :193–244, 1994.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Int., 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Int., 1991.

[Hol97] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5) :279–295, 1997.

[HR89] R. R. Howell and L. E. Rosier. Problems concerning fairness and temporal logic for conflict-free Petri nets. *Theoretical Computer Science*, 64(3) :305–329, 1989.

[HRY91] R. R. Howell, L. E. Rosier, and Hsu-Chun Yen. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science*, 82(2) :341–372, 1991.

[HS93] H. Hungar and B. Steffen. Local model checking for context-free processes. In *Proc. 20th Int. Coll. Automata, Languages, and Programming (ICALP'93), Lund, Sweden, July 1993*, volume 700 of *Lecture Notes in Computer Science*, pages 593–605. Springer, 1993.

[HT87] T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP'87), Karlsruhe, FRG, July 1987*, volume 267 of *Lecture Notes in Computer Science*, pages 269–279. Springer, 1987.

[IEC93] IEC (International Electrotechnical Commission). *IEC Standard 61131-3 : Programmable controllers - Part 3*, 1993.

[Jan90] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74(1) :71–93, 1990.

[Jan95] P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2) :281–301, 1995.

[Jan99] P. Jančar. A note on well quasi-orderings for powersets. *Information Processing Letters*, 72(5–6) :155–161, 1999.

[Jan00] P. Jančar. Decidability of bisimilarity for one-counter processes. *Information and Computation*, 158(1) :1–17, 2000.

[JEM99] P. Jančar, J. Esparza, and F. Moller. Petri nets and regular processes. *Journal of Computer and System Sciences*, 59(3) :476–503, 1999.

[JJ93] T. Jéron and C. Jard. Testing for unboundedness of fifo channels. *Theoretical Computer Science*, 113(1) :93–117, 1993.

[JK97] P. Jančar and A. Kučera. Bisimilarity of processes with finite-state systems. In F. Moller, editor, *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97), Bologna, Italy, July 1997*, UPMAIL Technical Report No. 148, pages 72–86, Uppsala, Sweden, 1997.

[JKM01] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. *Theoretical Computer Science*, 258(1–2) :409–433, 2001.

[JM95] P. Jančar and F. Moller. Checking regular properties of Petri nets. In *Proc. 6th Int. Conf. Theory of Concurrency (CONCUR'95), Philadelphia, PA, USA, Aug. 1995*, volume 962 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1995.

[JM96] L. Jategaonkar and A. R. Meyer. Deciding true concurrency equivalences on safe, finite nets. *Theoretical Computer Science*, 154(1) :107–143, 1996.

[JM99] P. Jančar and F. Moller. Techniques for decidability and undecidability of bisimilarity. In *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, Aug. 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1999.

[Jon75] N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1) :68–85, 1975.

[Kam68] J. A. W. Kamp. *Tense Logic and the theory of linear order.* PhD thesis, UCLA, Los Angeles, CA, USA, 1968.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2) :147–195, 1969.

[KM02] A. Kučera and R. Mayr. Weak bisimilarity between finite-state systems and BPA or normed BPP is decidable in polynomial time. *Theoretical Computer Science*, 270(1–2) :677–700, 2002.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proc. 5th Int. Conf. Computer Aided Verification (CAV'93), Elounda, Greece, June 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1993.

[Kou97] O. Kouchnarenko. *Sémantique des programmes récursifs-parallèles et méthodes pour leur analyse.* Thèse de Doctorat, Univ. Joseph Fourier-Grenoble I, France, February 1997.

[Koy89] R. Koymans. Specifing message passing systems requires extending temporal logic. In *Proc. Workshop Temporal Logic in Specification, Altrincham, UK, Apr. 1987*, volume 398 of *Lecture Notes in Computer Science*, pages 213–223. Springer, 1989.

[KP95] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symp. Logic in Computer Science (LICS'95), San Diego, CA, USA, June 1995*, pages 25–35. IEEE Comp. Soc. Press, 1995.

[Kri63] S. Kripke. Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 9 :67–96, 1963.

[Kru60] J. B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95 :210–225, 1960.

[Kru72] J. B. Kruskal. The theory of well-quasi-ordering : A frequently discovered concept. *Journal of Combinatorial Theory, Series A*, 13(3) :297–305, 1972.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes and three problems of equivalence. *Information and Computation*, 86(1) :43–68, 1990.

[KS96] O. Kouchnarenko and Ph. Schnoebelen. Modèles formels pour les programmes récursifs-parallèles. In *Proc. 8th Rencontres Francophones du Parallélisme (RenPar'96), Bordeaux, France, May 1996*, pages 85–88, 1996.

[KS97a] O. Kouchnarenko and Ph. Schnoebelen. A formal framework for the analysis of recursive-parallel programs. In *Proc. 4th Int. Conf. Parallel Computing Technologies (PaCT'97), Yaroslavl, Russia, Sep. 1997*, volume 1277 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 1997.

[KS97b] O. Kouchnarenko and Ph. Schnoebelen. A model for recursive-parallel programs. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, volume 5 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 1997.

[Kuč99] A. Kučera. Regularity of normed PA processes. *Information Processing Letters*, 72(1–2) :9–17, 1999.

[Kus00] D. Kuske. Emptiness is decidable for asynchronous cellular machines. In *Proc. 11th Int. Conf. Concurrency Theory (CONCUR'2000), University Park, PA, USA, Aug. 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 536–551. Springer, 2000.

[KV98] O. Kupferman and M. Y. Vardi. Relating linear and branching model checking. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98), Shelter Island, NY, USA, June 1998*, pages 304–326. Chapman & Hall, 1998.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2) :312–360, 2000.

[Lab98] A. Labroue. Conditions de vivacité dans les automates temporisés. Mémoire de DEA, DEA d'Informatique d'Orsay, July 1998. 60 pages.

[Lam80] L. Lamport. "Sometimes" is sometimes "Not Never". In *Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80), Las Vegas, NV, USA, Jan. 1980*, pages 174–185, 1980.

[Lam83] L. Lamport. What good is temporal logic ? In *Information Processing'83. Proc. IFIP 9th World Computer Congress, Sep. 1983, Paris, France*, pages 657–668. North-Holland, 1983.

[Lar94] F. Laroussinie. Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs. Thèse de Doctorat, I.N.P. de Grenoble, France, November 1994.

[LMS01] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Model checking CTL^+ and $FCTL$ is hard. In *Proc. 4th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2001), Genova, Italy, Apr. 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 318–331. Springer, 2001.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Principles of Programming Languages (POPL'85), New Orleans, LA, USA, Jan. 1985*, pages 97–107, 1985.

[LPS95] F. Laroussinie, S. Pinchinat, and Ph. Schnoebelen. Translations between modal logics of reactive systems. *Theoretical Computer Science*, 140(1) :53–71, 1995.

[LPY97] K. G. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1–2) :134–152, 1997.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Logics of Programs Workshop, Brooklyn, NY, USA, June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2) :303–324, 1995.

[LS00a] F. Laroussinie and Ph. Schnoebelen. Specification in CTL+Past for verification in CTL. *Information and Computation*, 156(1/2) :236–263, 2000.

[LS00b] F. Laroussinie and Ph. Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2000.

[LS00c] I. Lomazova and Ph. Schnoebelen. Some decidability results for nested Petri nets. In *Proc. 3rd Int. Andrei Ershov Memorial Conf. Perspectives of System Informatics (PSI'99), Novosibirsk, Russia, Jul. 1999*, volume 1755 of *Lecture Notes in Computer Science*, pages 208–220. Springer, 2000.

[LS00d] D. Lugiez and Ph. Schnoebelen. Decidable first-order transition logics for PA-processes. In *Proc. 27th Int. Coll. Automata, Languages, and Programming (ICALP'2000), Geneva, Switzerland, July 2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 342–353. Springer, 2000.

[LS01] A. Labroue and Ph. Schnoebelen. An automata-theoretic approach to the reachability analysis of RPPS systems. In *Proc. 8th Int. Workshop on Expressiveness in Concurrency (EXPRESS'2001), Aalborg, Denmark, Aug. 2001*, volume 52.1 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 2001.

[LS02] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1–2) :89–115, 2002.

[LST02] F. Laroussinie, Ph. Schnoebelen, and M. Turuani. On the expressivity and complexity of quantitative branching-time temporal logics. *Theoretical Computer Science*, 2002. To appear.

[Lyn77] N. Lynch. Log space recognition and translation of parenthesis languages. *Journal of the ACM*, 24(4) :583–590, 1977.

[May97a] R. Mayr. Combining Petri nets and PA-processes. In *Proc. 3rd Int. Symp. Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan, Sep. 1997*, volume 1281 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 1997.

[May97b] R. Mayr. Model checking PA-processes. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 1997.

[May97c] R. Mayr. Semantic reachability. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, volume 5 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 1997.

[May97d] R. Mayr. Tableaux methods for PA-processes. In *Proc. Int. Conf. Automated Reasoning with Analytical Tableaux and Related Methods (TABLEAUX'97), Pont-à-Mousson, France, May 1997*, volume 1227 of *Lecture Notes in Artificial Intelligence*, pages 276–290. Springer, 1997.

[May00a] R. Mayr. On the complexity of bisimulation problems for Basic Parallel Processes. In *Proc. 27th Int. Coll. Automata, Languages, and Programming (ICALP'2000), Geneva, Switzerland, July 2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 329–341. Springer, 2000.

[May00b] R. Mayr. Process rewrite systems. *Information and Computation*, 156(1/2) :264–286, 2000.

[May00c] R. Mayr. Undecidable problems in unreliable computations. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN'2000), Punta del Este, Uruguay, Apr. 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 377–386. Springer, 2000.

[May01] R. Mayr. Decidability of model checking with the temporal logic EF. *Theoretical Computer Science*, 256(1–2) :31–62, 2001.

[MC85] B. Mishra and E. M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38(2–3) :269–291, 1985.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 23(3) :267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall Int., 1989.

[MMKR94] L. E. Moser, P. M. Melliar-Smith, G. Kutty, and Y. S. Ramakrishna. Completeness and soundness of axiomatizations for temporal logics without next. *Fundamenta Informaticae*, 21(4) :257–305, 1994.

[Mol96] F. Moller. Infinite results. In *Proc. 7th Int. Conf. Concurrency Theory (CONCUR'96), Pisa, Italy, Aug. 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer, 1996.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Proc. REX School/Workshop, Noordwijkerhout, NL, May-June 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer, 1989.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proc. 9th ACM Symp. Principles of Distributed Computing (PODC'90), Quebec City, Canada, Aug. 1990*, pages 377–408, 1990.

[MS85] D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37(1) :51–75, 1985.

[NP92] M. Nivat and A. Podelski, editors. *Tree Automata and Languages*, volume 10 of *Studies in Computer Science and Artifical Intelligence*. North-Holland, 1992.

[NW63] C. St. J. A. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Philos. Soc.*, 59(4) :833–835, 1963.

[Old89] E.-R. Olderog. Strong bisimilarity on nets : a new concept for comparing net semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Proc. REX School/Workshop, Noordwijkerhout, NL, May-June 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 549–573. Springer, 1989.

[Old91] E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 1991.

[Pin93] S. Pinchinat. *Des Bisimulations pour la Sémantique des Systèmes Réactifs*. Thèse de Doctorat, I.N.P. de Grenoble, France, January 1993.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lect. Notes, Aarhus University, Aarhus, DK, 1981.

[PLS94] S. Pinchinat, F. Laroussinie, and Ph. Schnoebelen. Logical characterizations of truly concurrent bisimulation. Tech. Report 114, LIFIA-IMAG, Grenoble, March 1994. 16 pages.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA, Oct.-Nov. 1977*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1) :45–60, 1981.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Coll. Automata, Languages and Programming (ICALP'85), Nafplion, Greece, Jul. 1985*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 1985.

[Pnu86] A. Pnueli. Specification and development of reactive systems. In H.-J. Kugler, editor, *Information Processing'86, Proc. IFIP 10th World Computer Congress, Dublin, Ireland, Sep. 1986*, pages 845–858. North-Holland, 1986.

[Pri67] A. Prior. *Past, Present, and Future*. Clarendon Press, Oxford, 1967.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming, Turin, Italy, Apr. 1982*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[QS83] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems. A temporal logic to deal with fairness. *Acta Informatica*, 19(3) :195–220, 1983.

[Qui95] W. Quivrin-Pfister. *Des bisimulations de places pour la réduction des réseaux de Petri*. Thèse de Doctorat, I.N.P. de Grenoble, France, November 1995.

[Rab00] A. Rabinovich. Symbolic model checking for μ-calculus requires exponential time. *Theoretical Computer Science*, 243(1–2) :467–475, 2000.

[Rav97] L. Raviv. *LTL Symbolic Model Checking*. M.Sc. thesis, Weizmann Institute of Science, Rehovot, Israel, 1997.

[RM00] A. Rabinovich and S. Maoz. Why so many temporal logics climb up the trees ? In *Proc. 25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000), Bratislava, Slovakia, Aug. 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 629–639. Springer, 2000.

[RMD+92] Y. S. Ramakrishna, L. E. Moser, L. K. Dillon, P. M. Melliar-Smith, and G. Kutty. An automata-theoretic decision procedure for propositional temporal logic with Since and Until. *Fundamenta Informaticae*, 17(3) :271–282, 1992.

[RS00a] A. Rabinovich and Ph. Schnoebelen. BTL_2 and expressive completeness for $ECTL^+$. Research Report LSV-00-8, Lab. Specification and Verification, ENS de Cachan, Cachan, France, October 2000. Available at `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/`.

[RS00b] O. Rossi and Ph. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proc. 4th Int. Conf. Automation of Mixed Processes : Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany, Sept. 2000*, pages 177–182. Shaker Verlag, Aachen, Germany, 2000.

[SBB+99] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 1999.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3) :733–749, 1985.

[Sch85a] Ph. Schnoebelen. The semantics of concurrency in FP2. Research Report 558, LIFIA-IMAG, Grenoble, France, October 1985.

[Sch85b] Ph. Schnoebelen. *Sémantique du parallélisme en FP2*. Rapport de DEA, Univ. Grenoble, June 1985.

[Sch86a] Ph. Schnoebelen. About the implementation of FP2. Research Report 574, LIFIA-IMAG, Grenoble, France, January 1986.

[Sch86b] Ph. Schnoebelen. μ-FP2. A prototype interpreter for FP2. Research Report 573, LIFIA-IMAG, Grenoble, France, January 1986.

[Sch87] Ph. Schnoebelen. Rewriting techniques for the temporal analysis of communicating processes. In *Proc. Parallel Architectures and Languages Europe (PARLE'87), Eindhoven, NL, June 1987, vol. II : Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 402–419. Springer, 1987.

[Sch88] Ph. Schnoebelen. Refined compilation of pattern-matching for functional languages. *Science of Computer Programming*, 11(2) :133–159, 1988.

[Sch90a] Ph. Schnoebelen. Congruence properties of the process equivalence induced by temporal logic. Research Report 831-I, LIFIA-IMAG, Grenoble, France, October 1990.

[Sch90b] Ph. Schnoebelen. *Sémantique du parallélisme et logique temporelle. Application au langage FP2*. Thèse de Doctorat, I.N.P. de Grenoble, France, June 1990.

[Sch91] Ph. Schnoebelen. Experiments on processes with backtracking. In *Proc. 2nd Int. Conf. Theory of Concurrency (CONCUR'91), Amsterdam, NL, Aug. 1991*, volume 527 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 1991.

[Sch99] Ph. Schnoebelen. Decomposable regular languages and the shuffle operator. *EATCS Bull.*, 67 :283–289, 1999.

[Sch01] Ph. Schnoebelen. Bisimulation and other undecidable equivalences for lossy channel systems. In *Proc. 4th Int. Symp. Theoretical Aspects of Computer Software (TACS'2001), Sendai, Japan, Oct. 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2001.

[Sén00] G. Sénizergues. Complete formal systems for equivalence problems. *Theoretical Computer Science*, 231(2) :309–334, 2000.

[Sig00] M. Sighireanu. LOTOS NT user manual. release 2.1. Technical report, Projet VASY, Inria Rhône-Alpes, Montbonnot Saint-Martin, France, October 2000. Available via `http://www.inrialpes.fr/vasy/`.

[SJ89] Ph. Schnoebelen and Ph. Jorrand. Principles of FP2. Term algebras for specification of parallel machines. In J. W. de Bakker, editor, *Languages for Parallel Architectures : Design, Semantics, Implementation Models*, chapter 5, pages 223–273. Wiley, 1989.

[SP90] Ph. Schnoebelen and S. Pinchinat. On the weak adequacy of branching-time temporal logic. In *Proc. 3rd European Symp. on Programming (ESOP'90), Copenhagen, Denmark, May 1990*, volume 432 of *Lecture Notes in Computer Science*, pages 377–388. Springer, 1990.

[SR99] P.-Y. Schobbens and J.-F. Raskin. The logic of "initially" and "next" : Complete axiomatization and complexity. *Information Processing Letters*, 69(5) :221–225, 1999.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptative pattern matching. In *Proc. 19th Int. Coll. Automata, Languages, and Programming (ICALP'92), Vienna, Austria, July 1992*, number 623 in Lecture Notes in Computer Science, pages 247–260. Springer, 1992.

[SS87] P. Schaefer and Ph. Schnoebelen. Specification of a pipelined event-driven simulator using FP2. In *Proc. Parallel Architectures and Languages Europe (PARLE'87), Eindhoven, NL, June 1987, vol. I : Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 1987.

[SS00] Ph. Schnoebelen and N. Sidorova. Bisimulation and the reduction of Petri nets. In *Proc. 21st Int. Conf. Application and Theory of Petri Nets (ICATPN'2000), Aarhus, Denmark, June 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 2000.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1) :161–177, 1991.

[Tau89] D. Taubner. Representing CCS programs by finite predicate/transition nets. *Acta Informatica*, 27(6) :533–565, 1989.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1) :57–81, 1968.

[Vaa93] F. Vaandrager. Expressiveness results for process algebras. In *Semantics : Foundations and Applications, Proc. REX Workshop, Beekbergen, NL, June 1992*, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer, 1993.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. Theory of Computing (STOC'82), San Francisco, CA, May 1982*, pages 137–146, 1982.

[Var88] M. Y. Vardi. A temporal fixpoint calculus. In *Proc. 15th ACM Symp. Principles of Programming Languages (POPL'88), San Diego, CA, USA, Jan. 1988*, pages 250–259, 1988.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency : Structure Versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[Var98] M. Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer, 1998.

[VW86a] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. Logic in Computer Science (LICS'86), Cambridge, MA, USA, June 1986*, pages 332–344. IEEE Comp. Soc. Press, 1986.

[VW86b] M. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2) :183–221, 1986.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1) :1–37, 1994.

[Wil99] T. Wilke. CTL^+ is exponentially more succint than CTL. In *Proc. 19th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'99), Chennai, India, Dec. 1999*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 1999.

[WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths (extended abstract). In *Proc. 24th IEEE Symp. Foundations of Computer Science (FOCS'83), Tucson, AZ, USA, Nov. 1983*, pages 185–194, 1983.

[WZ78] C. H. West and P. Zafiropulo. Automated validation of a communications protocol : The CCITT X.21 recommendations. *IBM Journal of Research and Development*, 22(1) :60–71, 1978.

[YMW97] J. Yang, A. K. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. *ACM Transactions on Programming Languages and Systems*, 19(2) :386–412, 1997.

[Yov97] S. Yovine. Kronos : A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1–2) :123–133, 1997.