ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

THÈSE

présentée
pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Spécialité INFORMATIQUE

PAR

Raphaël MEYER

Contributions à l'étude des logiques temporelles sur les traces

Soutenue le 19 novembre 1999 devant le jury composé de :

MM.	Jean BERSTEL	Président
	Alain FINKEL	Examinateur
	Paul GASTIN	Examinateur
	Yves MÉTIVIER	Rapporteur
	Antoine PETIT	Directeur de thèse
	P. S. THIAGARAJAN	Rapporteur

Laboratoire **S**pécification et **V**érification – UMR 8643 CNRS;
ENS de Cachan; 61, Avenue du Président Wilson; 94235 CACHAN Cedex

Je tiens à remercier chaleureusement Antoine Petit, mon directeur de thèse, pour son soutien et son exigence.

Je remercie les membres du jury d'avoir accepté de participer au "final" de cette thèse, et notamment les rapporteurs.

Je remercie également tous les membres et personnels du LSV de l'ambiance de travail qu'ils ont su créer.

Enfin, un très grand merci à Carina.

Table des matières

Chapitre 1

Introduction

Depuis l'avènement de réseaux informatiques à l'échelle planétaire, la notion d'accès à l'information a considérablement évoluée. Aujourd'hui, la révolution du travail qu'engendre l'informatisation permet aux entreprises de disséminer leurs employés dans des lieux géographiquement éloignés les uns des autres, sans que cela n'entrave la possibilité qu'à chacun d'accéder à l'information dont il a besoin : il est parfois plus rapide de demander, à l'aide des réseaux informatiques, des renseignements à un collègue travaillant de l'autre côté de la planète que de descendre trois étages pour trouver ces mêmes renseignements... Cette réalité s'accompagne de nouveaux défis, tant pour l'informatique pratique que pour l'informatique théorique.

Fiabilité des systèmes informatiques La *fiabilité* et la *sûreté* des systèmes constituent désormais un domaine d'investigations primordial pour le secteur de l'informatique, car les enjeux qui en découlent sont d'une importance croissante, tant sur le plan humain que sur le plan financier : en effet, de plus en plus de systèmes critiques reposent sur des programmes informatiques. C'est le cas dans les domaines du nucléaire, de l'avionique, des télécommunications, de l'aéronautique et des systèmes bancaires entre autres... Lorsque la fiabilité de tels systèmes devient un problème crucial, il est nécessaire d'être capable, avant la mise en service de ces systèmes, d'affirmer que leur comportement satisfera certaines exigences. L'utilisation de *méthodes formelles* pour la validation des programmes est une des réponses possibles à de tels problèmes.

L'essence des méthodes formelles consiste à représenter le système informatique étudié par un modèle mathématique, et à l'analyser de ce point de vue. D'une part, on cherche des modèles assez *expressifs* pour pouvoir *spécifier* toutes les propriétés nécessaires à la *validation* du système; d'autre part on doit limiter cette expressivité afin de réduire la *complexité* mathématique inhérente à l'analyse du système. En particulier, lorsqu'on étudie des systèmes distribués (ce qui est de plus en plus fréquent puisque la plupart des systèmes réels sont constitués de plusieurs parties qui interagissent), le nombre d'états du système croît exponentiellement avec le nombre de ses composantes. Nous reviendrons sur ce problème après avoir expliqué plus précisémment en quoi consistent les principaux problèmes de vérification qui nous préoccupent.

La vérification La *vérification* est un ensemble de méthodes formelles qui reposent sur l'analyse exhaustive de l'espace des états d'un système. Elle consiste à évaluer, de manière plus ou moins automatique, l'ensemble des comportements d'un programme vis à vis d'une propriété donnée. En général, cette propriété est exprimée dans un formalisme particulier appelé *logique temporelle*. Il s'agit d'un formalisme qui apparaît naturellement lorsque que l'on cherche à exprimer des propriétés décrivant l'évolution d'un système au cours du temps.

Deux des problèmes majeurs de la vérification sont :

- *La Satisfaisabilité* : étant donnée une formule de spécification ϕ exprimée en logique temporelle, existe-t-il un modèle M (dans une classe donnée) qui satisfait ϕ?

- *Le Model-Checking* : étant donnés un modèle M et une formule ϕ, est-ce que M satisfait ϕ?

Dans la suite de cette thèse nous reviendrons souvent sur ces deux problèmes, et notamment sur leur complexité.

Entrelacements et explosion combinatoire Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à la vérification des systèmes distribués. Une manière simple et naturelle d'étudier un tel système est de considérer un observateur extérieur séquentiel. De son point de vue, toutes les actions (ou tous les débuts et fins d'actions) sont exécutées à des dates distinctes et sont donc ordonnées - on parle alors d'ordonnancement temporel des actions. Ce point de vue, appelé *sémantique des entrelacements* a été largement développé pour les systèmes distribués et s'est souvent avéré efficace. En particulier, il permet de ramener un problème posé sur un système distribué à un problème sur un système séquentiel, et par conséquent d'appliquer avec succès toutes les méthodes usuelles de ce domaine.

Néanmoins, cette sémantique présente un inconvénient majeur pour de nombreuses classes de systèmes distribués. Supposons en effet que l'on étudie un ensemble constitué de n processus séquentiels représenté chacun par un automate à k états. Si on exécute en parallèle ces processus, alors le processus concurrent résultant possède k^n états. Sa taille croît ainsi exponentiellement avec le nombre de composants. Cette situation est connue sous le nom d'*explosion (combinatoire) de l'espace des états* et constitue évidemment un frein important à la vérification automatique du système distribué étudié.

Méthodes d'ordre partiel Penchons nous plus avant sur la structure des systèmes distribués afin de dégager des solutions au problème de l'explosion combinatoire de l'espace des états. Lorsque l'on étudie les systèmes distribués, on est souvent amené à considérer un ensemble d'agents dont les comportements individuels sont séquentiels mais qui sont coordonnés par divers mécanismes de communication. Dans un tel système, la plupart des tâches effectuées sont internes, au sens où elles ne concernent qu'un unique agent; seules certaines tâches particulières correspondent à des communications entre les agents. Chaque agent suit donc une alternance de phases internes (pendant lesquelles il ne communique avec aucun autre) et de phases de communications (pendant lesquelles il échange des informations avec d'autres agents). Ainsi, lorsque deux agents distincts sont dans des phases internes, les tâches que l'un d'entre eux effectue sont causalement indépendantes de celles qu'effectue l'autre agent. Dans un tel système, de nombreuses exécutions ne différeront que par l'ordonnancement d'actions internes indépendantes. Dans la sémantique des entrelacements, de telles exécutions, dites équivalentes, sont considérées séparément. Les *méthodes d'ordre partiel* visent à regrouper ces exécutions équivalentes par classe d'équivalence (le nom de ces méthodes vient du fait que des exécutions équivalentes sont des linéarisations d'un même ordre partiel) et à n'explorer qu'une seule séquence, dite représentative, par classe pour s'assurer de la satisfaction de la propriété étudiée. Bien entendu, une telle méthode ne peut être utilisée que si la propriété en question est *robuste*, c'est-à-dire si elle est vérifiée par une séquence d'actions dès qu'elle l'est par une séquence équivalente. C'est par exemple le cas lorsque l'on recherche des situations de blocage dans un système distribué.

Les travaux pionniers de ces méthodes d'ordre partiel sont dus à Overman [36] qui dès 1981 proposait de réduire l'espace des états engendré par la mise en parallèle de processus séquentiels en utilisant des informations concernant la concurrence de certaines actions provenant des différents processus. Ses résultats ne s'appliquent néanmoins que dans des cas très restreints, tant en ce qui concerne les modèles de parallélisme qu'en ce qui concerne les propriétés vérifiables.

En 1988, une partie des problèmes rencontrés par Overman ont été résolus par Valmari, grâce à une méthode dite des ensembles "obstinés" (*stubborn sets*) [47]. Cette méthode permet de réduire l'exploration de l'espace des états grâce à des ensembles calculés au cours de cette exploration. Elle ne s'applique alors qu'à certaines propriétés de type blocages (*deadlocks*), mais Valmari l'a ensuite étendue à la vérification de propriétés plus complexes, et notamment à des propriétés exprimables dans une logique temporelle semblable à la logique temporelle linéaire LTL de Pnueli [39], mais privée de l'opérateur "next" [48, 49].

Les travaux de Valmari ont été améliorés et généralisés par Godefroid et Wolper [18, 19] qui ont proposé les notions d'ensembles "dormants" et "persistants" (*sleep sets* et *persistent sets*). Ces techniques ont notamment été implémentées au sein de l'outil de vérification SPIN [17].

Parallèlement à cette voie, Katz et Peled ont développé une logique temporelle nommée ISTL (*Interleaving Set Temporal Logic*) qui intègre le fait que des séquences d'actions qui ne diffèrent que par l'ordonnancement d'actions concurrentes sont équivalentes [22]. Cette logique est particulièrement expressive, mais seuls certains fragments en sont décidables [1].

Les modèles concurrents Dans certains cas de la vérification automatique, les méthodes d'ordre partiel permettent un gain indéniable en pratique, car les réductions qui en résultent, aussi bien en temps qu'en espace, sont souvent importantes. Néanmoins, ces méthodes ne sont pas tout à fait satisfaisantes car elles ne font que palier les défauts des méthodes classiques, sans apporter une solution réellement adaptée au problème des systèmes concurrents. Notamment, elles nécessitent de n'avoir à vérifier que des propriétés robustes, et le fait de décider si une propriété est ou n'est pas robuste peut s'avérer être un problème difficile. C'est pourquoi on est naturellement amené à étudier des modèles présentant un parallélisme intrinsèque et à chercher des logiques temporelles qui puissent leur être appliquées.

Les traces Dans le cadre de cette thèse, on s'intéresse en particulier à un modèle mathématique présentant ce parallélisme intrinsèque : les traces. Il s'agit d'un modèle introduit en 1977 par Mazurkiewicz [28], et qui a depuis été largement étudié [8, 9]. Les traces permettent de s'affranchir de la distinction entre différentes séquences d'actions équivalentes. Notre but est d'étudier la possibilité de développer des logiques temporelles expressives et efficaces, dont la sémantique est définie directement sur les traces. Plusieurs logiques temporelles sur les traces ont déjà été proposées, notamment TLPO [10], TrPTL [44], et LTrL [46]. Nous présenterons un panorama détaillé de ces logiques au Chapitre 3. Ces logiques se répartissent dans deux catégories distinctes : logiques *locales* et logiques *globales*. Les logiques locales possèdent des opérateurs temporels liés à des processus particuliers intervenants dans le système, alors que les logiques globales décrivent des propriétés des états globaux du système étudié. Nous verrons par la suite que cette distinction est liée à l'expressivité et à la complexité de ces logiques : les logiques locales sont en général moins expressives que les logiques globales, mais leur complexité est plus faible. C'est donc dans ce cadre que s'inscrit le travail de cette thèse, que nous détaillons maintenant.

Plan de la thèse Dans le chapitre suivant, nous explicitons les bases théoriques de la théorie des traces, et nous rappelons notamment quelques résultats concernants les liens entre les langages de traces et les logiques du premier ordre et monadique du second ordre.

Au chapitre 3, nous dressons un aperçu de l'historique des relations entre les traces et les logiques temporelles, et nous donnons quelques éléments de classification de ces logiques, ainsi que des critères que devrait satisfaire une logique temporelle "idéale" pour les traces.

Dans le chapitre 4, nous étudions plus en détail une logique particulière, TrPTL, introduite par Thiagarajan [44], et nous prouvons un résultat de décomposition des formules dans le contexte d'un alphabet de dépendance non-connexe. Ce résultat peut être mis en parallèle avec le Théorème de Mezei pour les langages de mots reconnaissables [2]. Dans notre contexte, ce résultat montre que lorsque l'on parle d'un système composé de plusieurs sous-systèmes indépendants (i.e. sans communications entre eux), alors toute spécification écrite dans la logique TrPTL concernant l'ensemble du système peut être réécrite comme une combinaison booléenne de spécifications ne concernant qu'un sous-système à la fois.

Dans le chapitre 5, nous nous intéressons à une autre logique temporelle, LTrL, proposée par Thiagarajan et Walukiewicz [46], et nous exhibons pour cette logique un algorithme de décision *modulaire*, mais de complexité non-élémentaire. Le caractère modulaire de l'algorithme signifie que cette procédure s'effectue de façon inductive sur la structure des formules. Cet algorithme permet de décider de la *satisfaisabilité* d'une formule quelconque de LTrL, et nous verrons dans ce chapitre que toute la complexité d'une telle procédure provient à la fois de la négation et de l'utilisation de l'opérateur *until* (noté $\mathcal{U}$), où une formule du type $\alpha \, \mathcal{U} \, \beta$ est satisfaite si β l'est dans le futur, et si entretemps α est toujours satisfaite.

Au chapitre 6, nous abordons un aspect plus pratique de la vérification utilisant des logiques temporelles sur les traces en nous intéressant à un problème de model-checking dans le cadre restreint des graphes de signaux de transitions. Il s'agit d'une classe de réseaux de Petri qui est utilisée pour la spécification de certaines architectures matérielles asynchrones. Nous montrons en particulier que dans ce cadre le problème du model-checking admet une solution de faible complexité pour les formules exprimées dans la logique temporelle LTrL.

Enfin, le chapitre 7 est consacré aux conclusions que l'on peut tirer de cette étude, ainsi qu'aux perspectives de recherches qui restent à explorer.

Chapitre 2

Préliminaires

On donne dans ce chapitre les principales notations utilisées par la suite, ainsi que les définitions essentielles qui nous serviront dans les chapitres ultérieurs. La plupart des résultats énoncés dans ce chapitre se trouve dans le "Book of Traces", ouvrage collectif édité par Diekert et Rozenberg [9], ainsi que dans le chapitre *Partial Commutation and Traces* de Diekert et Métivier du "Handbook of Formal Languages" [8].

2.1 Traces de Mazurkiewicz

Le point de départ de la théorie des traces, initiée par Mazurkiewicz [28], est la notion d'*alphabet de dépendance.*

2.1.1 Alphabet de dépendance

Soit Σ un ensemble fini, dénommé *alphabet*, et dont les éléments, appelés *actions* seront notés $a, b, c, \ldots$. L'idée intuitive de Mazurkiewicz est de doter Σ d'informations complémentaires concernant la façon dont certaines actions peuvent, ou ne peuvent pas, être exécutées en parallèle.

Un *alphabet de dépendance* est un couple (Σ, D) où Σ est un alphabet et $D \subseteq \Sigma \times \Sigma$ une relation réflexive et symétrique: pour tout couple d'actions (a, b), si $(a, b) \in D$ alors $(b, a) \in D$, et pour toute action a de Σ, $(a, a) \in D$. La relation complémentaire $I = \Sigma \times \Sigma \setminus D$ est appelée la *relation d'indépendance.* Intuitivement, les actions a et b sont indépendantes (donc peuvent être exécutées en parallèle) si et seulement si $(a, b) \in I$. Cette relation I est fixe, elle n'évolue pas dans le temps : deux actions indépendantes le resteront, de même que deux actions dépendantes. Cette invariance de I constitue une particularité importante de la théorie des traces.

2.1.2 Alphabet distribué

Une notion proche de celle d'alphabet de dépendance est celle d'alphabet distribué. Cette dernière notion provient d'une vision plus proche de l'implémentation des systèmes distribués.

Considérons un ensemble fini $\mathcal{P}$ de processus. Un *alphabet distribué* est une famille $\{\Sigma_p\}_{p\in\mathcal{P}}$ où, pour tout $p \in \mathcal{P}$, Σ_p est un alphabet non-vide. Cet alphabet Σ_p représente l'ensemble des actions que peut effectuer le processus p. A tout alphabet distribué $\{\Sigma_p\}_{p\in\mathcal{P}}$ on peut associer naturellement l'ensemble $\Sigma_\mathcal{P} = \cup_{p\in\mathcal{P}}\Sigma_p$ de toutes les actions possibles du système de processus. De plus, à toute action $a \in \Sigma_\mathcal{P}$ on associe l'ensemble $pr(a) = \{p \in \mathcal{P} \mid a \in \Sigma_p\}$ des processus qui participent à l'exécution de cette action. On peut alors définir une relation de dépendance $D_\mathcal{P}$ sur l'alphabet $\Sigma_\mathcal{P}$ de la façon suivante :

$$\forall a, b \in \Sigma_\mathcal{P}, (a, b) \in D_\mathcal{P} \text{ ssi } pr(a) \cap pr(b) \neq \emptyset$$

Autrement dit, deux actions sont indépendantes si et seulement si les processus nécessaires à leurs exécutions forment des ensembles disjoints. Il est clair qu'alors ces actions peuvent être exécutées en parallèle, puisqu'elles utilisent des ressources distinctes.

Ainsi, tout alphabet distribué $\{\Sigma_p\}_{p\in\mathcal{P}}$ induit naturellement un alphabet de dépendance $\{\Sigma_\mathcal{P}, D_\mathcal{P}\}$. On comprend sur cette construction simple que la notion d'alphabet de dépendance est légèrement plus abstraite que celle d'alphabet distribué, au sens où dans la première on "oublie" la provenance des actions pour ne garder que les relations existantes entre ces actions.

Inversement, partant d'un alphabet de dépendance donné (Σ, D), il existe de nombreuses façons de l'implémenter en un alphabet distribué. Une méthode classique et naturelle consiste à considérer un processus pour chaque D-clique maximale de (Σ, D), une D-clique étant un sous-ensemble non-vide p de Σ vérifiant $p \times p \subseteq D$. Si l'on note $\mathcal{P}$ l'ensemble des D-cliques maximales de (Σ, D), et si pour tout p de $\mathcal{P}$ on pose $\Sigma_p = p$, alors l'alphabet distribué $\{\Sigma_p\}_{p\in\mathcal{P}}$ implémente (Σ, D) au sens où l'alphabet de dépendance naturellement associé à $\{\Sigma_p\}_{p\in\mathcal{P}}$ est (Σ, D).

Cette approche est illustrée par l'exemple suivant :

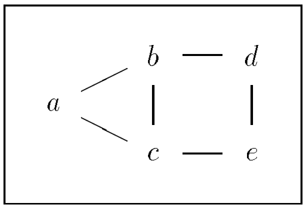

FIG. 2.1 – *Graphe de la relation D sur l'alphabet Σ*

Exemple 1 *Soit l'alphabet de dépendance (Σ, D), où $\Sigma = \{a, b, c, d, e\}$ et où la relation D est illustrée sur la Figure 2.1. En considérant les D-cliques maximales de cet alphabet, on lui associe l'alphabet distribué $\{\{a, b, c\}, \{b, d\}, \{d, e\}, \{c, e\}\}$.*

2.1.3 Graphes de dépendances

On s'intéresse maintenant à un graphe de dépendance (Σ, D) fixé, et à une classe particulière de graphes étiquetés par Σ.

On appelle *ensemble partiellement ordonné étiqueté par* Σ un triplet $F = (E, \leq, \lambda)$ tel que E soit un ensemble, $\leq$ une relation d'*ordre partiel* sur E, et λ une fonction de E vers Σ. Rappelons qu'un *ordre partiel* $\leq$ sur E est une relation $\leq \subseteq E \times E$ transitive, réflexive, et antisymétrique. Contrairement à la définition d'un ordre total, on n'exige pas que tous les couples de $E \times E$ soit ordonnés par $\leq$. Les éléments de E sont appelés les *événements* de F.

Pour des raisons de commodité, on notera en abrégé $\Sigma-$poset (*p*artially *o*rdered *set*) la notion d'ensemble partiellement ordonné étiqueté par Σ.

Si l'on dispose d'un $\Sigma-$poset $F = (E, \leq, \lambda)$, la relation d'ordre partiel $\leq$ induit une *relation de couverture* $\lessdot \subseteq E \times E$ définie par : $\forall e, e' \in E$, $e \lessdot e'$ ssi $e \leq e'$, $e \neq e'$ et pour tout $e'' \in E$, si $e \leq e'' \leq e'$ alors $e'' = e$ ou $e'' = e'$. En d'autres termes, $e \lessdot e'$ ssi e est strictement "plus petit" que e' et s'il n'y a pas d'autre événement de E entre les deux vis-à-vis de la relation d'ordre partiel $\leq$.

Enfin, pour tout sous-ensemble d'événements $X \subseteq E$, on note $\downarrow\!X$ et on appelle *passé de* X l'ensemble $\{y \in E \mid \exists x \in X, y \leq x\}$, i.e. l'ensemble des événements de E dominés par des événements de X. Si l'on interprète intuitivement la relation $\leq$ comme une relation de causalité entre les événements de E, alors $\downarrow X$ correspond à l'ensemble des événements antérieurs à X, donc au passé (causal) de X. Si X est un singleton $\{x\}$, on notera $\downarrow x$ l'ensemble $\downarrow\{x\}$. On a alors $\downarrow X = \bigcup_{x \in X} \downarrow x$.

Définition 1 *Un* graphe de dépendance *sur* (Σ, D) *est un* $\Sigma-$*poset* $F = (E, \leq, \lambda)$ *satisfaisant les conditions suivantes :*

(GD1) *E est au plus dénombrable.*

(GD2) *Pour tout $e \in E$, l'ensemble $\downarrow e$ est fini.*

(GD3) *Pour tous $e, e' \in E$, si $e \lessdot e'$ alors $(\lambda(e), \lambda(e')) \in D$.*

(GD4) *Pour tous $e, e' \in E$, si $(\lambda(e), \lambda(e')) \in D$ alors $e \leq e'$ ou $e' \leq e$.*

Les sommets de ce graphe sont les événements de E, étiquetés par leur image dans Σ à travers la fonction λ, et les arêtes de ce graphe sont décrites par la relation de causalité $\leq$.

Remarque 1 *Si $F = (E, \leq, \lambda)$ et $F' = (E', \leq', \lambda')$ sont deux graphes de dépendance sur (Σ, D), on note $F \equiv F'$ s'il existe un* isomorphisme *θ entre F et F', i.e. :*

- $\theta(E) = E'$
- $\forall e \in E,\ \lambda(e) = \lambda'(\theta(e))$
- $\forall e, e' \in E,\ e \leq e'$ *ssi* $\theta(e) \leq' \theta(e')$

La relation $\equiv$ est une relation d'équivalence sur l'ensemble des graphes de dépendance, pour un alphabet (Σ, D) fixé.

On note $\mathbf{G}(\Sigma, D)$ l'ensemble des classes d'équivalences de graphes de dépendance sur (Σ, D) (i.e. l'ensemble des graphes de dépendance définis à un isomorphisme près). Intuitivement, la condition $(GD2)$ de la Définition 1 permet d'éviter les situations dans lesquelles un sommet du graphe ne serait atteignable que par une suite infinie d'arêtes. Les conditions $(GD3)$ et $(GD4)$ stipulent qu'un graphe de dépendance doit "respecter" la relation de dépendance D : si deux sommets sont ordonnés, sans qu'il y ait de sommet intermédiaire entre eux, alors leurs étiquettes sont dépendantes; et réciproquement,

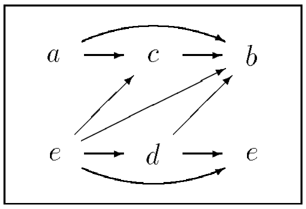

FIG. 2.2 – *Graphe de dépendance fini sur* (Σ, D)

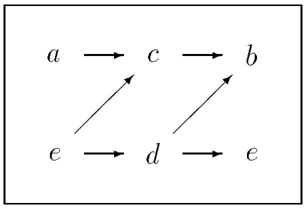

FIG. 2.3 – *Diagramme de Hasse du graphe de dépendance de la Fig. 2.2*

si les étiquettes de deux sommets sont dépendantes, alors ces sommets sont ordonnés.

Remarque 2 *Si* $F = (E, \leq, \lambda)$ *est un graphe de dépendance, son* diagramme de Hasse *est le graphe obtenu à partir de* F *en retirant toutes les arêtes qui peuvent se déduire par transitivité d'autres arêtes. Autrement dit, dans un diagramme de Hasse, on ne considère que les arêtes décrites par la relation de couverture du graphe de dépendance.*

Exemple 2 *Soit* (Σ, D) *l'alphabet de dépendance de l'Exemple 1, dont le graphe est représenté sur la Fig. 2.1. La Fig. 2.2 montre un graphe de dépendance fini sur cet alphabet, et la Fig. 2.3 montre le diagramme de Hasse associé à ce graphe.*

Une notion primordiale sur les graphes de dépendance est celle de configuration. Une *configuration* d'un graphe de dépendance $F = (E, \leq, \lambda)$ est un sous-ensemble fini $c \subseteq E$ de sommets clos par le passé, i.e. vérifiant $c = \downarrow c$. Cette notion correspond pour les graphes de dépendance au concept de préfixe : en effet, toute configuration c d'un graphe de dépendance F induit un nouveau graphe de dépendance $F' = (c, \leq_c, \lambda_c)$, où $\leq_c$ et λ_c sont respectivement les restrictions à $c \times c$ de $\leq$ et à c de λ. Pour un graphe de dépendance F, on note C_F l'ensemble des configurations de F.

Si c et c' sont deux configurations de F, on note $c \xrightarrow{a} c'$ lorsqu'il existe $e \in E \setminus c$ tel que $c' = c \cup \{e\}$ et $\lambda(e) = a$. On dit alors qu'il existe une transition étiquetée par a de c vers c'.

Remarque 3 *Pour tout graphe de dépendance F, l'ensemble vide $\emptyset$ est une configuration de F, i.e. $\emptyset \in C_F$. Cette configuration particulière est souvent appelée* configuration initiale.

Remarquons que, pour un graphe de dépendance général F, l'ensemble C_F des configurations de F n'est muni que d'une relation d'ordre partiel (pour l'inclusion). Ainsi, tous les "préfixes" de F ne sont pas ordonnés, contrairement à ce qui se passe par exemple dans le cas des mots, où les préfixes d'un mot donné sont totalement ordonnés par inclusion.

2.1.4 Classes d'équivalence de mots

On s'intéresse maintenant à une notion a priori fort différente de celle de graphe de dépendances, mais qui, on le verra, lui est en fait équivalente. On part à nouveau d'un alphabet de dépendance (Σ, D) fixé. Les notations Σ^* et Σ^ω désignent respectivement l'ensemble des mots finis et infinis sur l'alphabet Σ. On note $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ l'ensemble des mots finis ou infinis sur Σ. Pour tout mot u de Σ^∞ et tout sous-alphabet $\Gamma \subseteq \Sigma$, on note $u_{|\Gamma}$ le mot de Σ^∞ obtenu à partir de u en effaçant toutes les lettres de $\Sigma \setminus \Gamma$. Plus formellement, on définit un morphisme $\pi_\Gamma : \Sigma^\infty \to \Gamma^\infty$ en posant, pour tout $a \in \Gamma$, $\pi_\Gamma(a) = a$, et pour tout $a \in \Sigma \setminus \Gamma$, $\pi_\Gamma(a) = \varepsilon$, où ε désigne le mot vide. Alors $u_{|\Gamma} = \pi_\Gamma(u)$.

Alors la relation d'indépendance $I = \Sigma \times \Sigma \setminus D$ permet de définir une relation d' équivalence $\sim_I$ sur Σ^∞, de la façon suivante :

$$\forall u, v \in \Sigma^\infty,\ u \sim_I v \text{ ssi } u_{|\Gamma} = v_{|\Gamma} \text{ pour toute } D\text{-clique } \Gamma$$

.

Définition 2 *Une* trace *sur l'alphabet de dépendance (Σ, D) est une classe d'équivalence de mots de Σ^∞ pour la relation d'équivalence $\sim_I$ induite par I.*

Remarque 4 *Si $u = rabs$ et $v = rbas$ avec $r \in \Sigma^*$, $s \in \Sigma^\infty$ et $(a,b) \in I$, alors $u \sim_I v$. Plus généralement, si on peut obtenir v par une suite finie de permutations de paires de lettres adjacentes et indépendantes dans u, alors u et v sont équivalents pour $\sim_I$. Pour les mots finis, on peut même définir $\sim_I$ directement comme la fermeture transitive de la congruence induite par la relation $\{(ab,ba) \mid (a,b) \in I\}$ sur Σ^*.*

Exemple 3 *Soit (Σ, D) l'alphabet de dépendance de la Figure 2.1. Alors les mots suivants sont équivalents :*

$$abcddea \sim_I abdcdea \sim_I abddcea \sim_I abddcae$$

.

Si $u \in \Sigma^\infty$ est un mot, on note généralement $[u]$ la trace correspondante, i.e. la classe d'équivalence de u dans Σ^∞. Par ailleurs, on note $\mathbf{R}(\Sigma, D)$ l'ensemble des traces sur l'alphabet de dépendance (Σ, D), c'est-à-dire :

$$\mathbf{R}(\Sigma, D) = \Sigma^\infty / \sim_I$$

Remarquons que si u est un mot fini, alors tous les mots $v \in [u]$ le sont également et ont la même longueur que u. Ceci implique que si u est un mot infini, alors tous les mots $v \in [u]$ sont également infinis. On peut donc parler de *traces finies* et de *traces infinies*. Plus précisément, notons φ l'application canonique de Σ^∞ dans $\mathbf{R}(\Sigma, D)$ qui à tout mot u associe sa classe d'équivalence $[u]$:

$$\Sigma^\infty \xrightarrow{\varphi} \mathbf{R}(\Sigma, D)$$

Alors l'ensemble des traces finies $\mathbf{M}(\Sigma, D)$ est l'image par φ de l'ensemble des mots finis Σ^* :

$$\mathbf{M}(\Sigma, D) = \varphi(\Sigma^*)$$

L'ensemble $\mathbf{R}(\Sigma, D)$ des traces finies et infinies est appelé ensemble des traces réelles sur (Σ, D).

Par ailleurs, pour toute trace $[u] \in \mathbf{R}(\Sigma, D)$ on peut définir l'alphabet de $[u]$, noté $\mathrm{alph}([u])$, comme l'ensemble des lettres de Σ ayant au moins une occurrence dans u. Cette définition ne dépend pas du représentant de la trace choisi, car si $[u] = [v]$, alors pour toute lettre $a \in \Sigma$, $\pi_a(u) = \pi_a(v)$ et on a précisément

$$\mathrm{alph}([u]) = \{a \in \Sigma \mid \pi_a(u) \neq \varepsilon\}.$$

L'ensemble Σ^* possède une structure algébrique particulière : c'est un *monoïde*, i.e. il est fermé par le produit de concaténation :

$$\forall u, v \in \Sigma^*, uv \in \Sigma^*$$

L'ensemble des traces finies $\mathbf{M}(\Sigma, D)$ sur l'alphabet (Σ, D) possède une structure identique : c'est également un monoïde pour le produit de concaténation que l'on définit comme suit : si t et r sont deux traces finies, il existe deux mots finis u et v tels que $t = [u]$ et $r = [v]$; on pose alors $tr = [uv]$. Cette définition ne dépend pas du choix de u et v car si $u' \sim_I u$ et si $v' \sim_I v$, alors $u'v' \sim_I uv$ donc $[u'v'] = [uv]$. Cette définition de la concaténation dans $\mathbf{M}(\Sigma, D)$ implique immédiatement que φ est un *morphisme de monoïdes*, i.e. :

$$\forall u, v \in \Sigma^*, \varphi(uv) = \varphi(u)\varphi(v)$$

L'étude des traces a donné lieu à de nombreux résultats, qui sont parfois des généralisations de résultats déjà connus sur les mots. Néanmoins, de telles généralisations sont souvent plus riches que les résultats correspondants sur les mots. C'est le cas en particulier du Lemme de Levi, qui sur les mots exprime le fait que, lorsque l'on a une égalité du type $uv = xy$, et si u est plus court que x, alors il existe un mot r qui est à la fois un suffixe de x et un préfixe de v. On a alors l'égalité $uv = xy = ury$. Dans le cas des traces, ce Lemme se généralise de façon plus symétrique, comme suit :

Théorème 1 (Lemme de Levi [6]) *Soient des traces finies* $[u]$,$[v]$,$[x]$,$[y]$ *telles que* $[u][v] = [x][y]$. *Alors il existe des traces* $[z_1]$,$[z_2]$,$[z_3]$,$[z_4]$ *telles que*

$$[u] = [z_1][z_2], [v] = [z_3][z_4], [x] = [z_1][z_3], [y] = [z_2][z_4]$$

et telles que $\mathrm{alph}([z_2]) \times \mathrm{alph}([z_3]) \subseteq I$.

Comme pour les mots, on peut définir une notion de *préfixe* sur les traces : soient $[u], [v] \in \mathbf{R}(\Sigma, D)$, alors $[u]$ est un préfixe de $[v]$ ssi il existe $[w] \in \mathbf{R}(\Sigma, D)$ tel que $[v] = [u][w] = [uw]$. On note alors $[u] \preceq [v]$. Pour toute trace $[u]$ on note $pr([u])$ l'ensemble des traces préfixes de u. On peut alors définir les vues partielles d'une trace finie : pour tout $a \in \Sigma$ et tout $[u] \in \mathbf{R}(\Sigma, D)$, la vue partielle de $[u]$ par a est

$$\partial_a([u]) = Min_{\preceq}\{[v] \mid \exists [w], [u] = [v][w] \text{ et } a \notin \mathrm{alph}([w])\}$$

où alph($[w]$) désigne l'ensemble des lettres de Σ ayant au moins une occurrence dans $[w]$. La trace $\partial_a([u])$ est toujours définie sans ambiguïté. En effet, l'existence de $\partial_a([u])$ vient du fait que $[u] = [u][\varepsilon]$ et $\mathrm{alph}([\varepsilon]) \times \{a\} \subseteq I$ et l'unicité vient de ce que si on avait $\partial_a([u]) \supseteq \{[v],[v']\}$ alors $[v][w] = [v'][w']$ et en appliquant le Lemme de Levi on obtiendrait facilement une contradiction.

Cette notion de vue partielle peut être généralisée facilement à un sous-alphabet quelconque de Σ : pour tout $[u] \in \mathbf{R}(\Sigma, D)$ et pour tout $\Gamma \subseteq \Sigma$, on définit

$$\partial_\Gamma([u]) = Min_\preceq\{[v] \mid \exists [w], [u] = [v][w] \text{ et } \mathrm{alph}([w]) \cap \Gamma = \emptyset\}.$$

Si l'on part d'un alphabet distribué $\{\Sigma_p\}_{p\in\mathcal{P}}$, alors pour toute trace $[u]$, la trace préfixe $\partial_{\Sigma_P}([u])$ représente la meilleure connaissance qu'ait le processus p d'une exécution produisant la trace $[u]$, puisque le suffixe correspondant dans $[u]$ ne comporte aucune action de Σ_p, i.e. son exécution ne fait jamais appel au processus p. Ce processus ne peut donc pas *distinguer* deux traces $[u]$ et $[v]$ ayant le même préfixe $\partial_{\Sigma_p}([u]) = \partial_{\Sigma_p}([v])$. C'est la raison pour laquelle ces préfixes sont nommés vues partielles.

Une autre notion intéressante liée à celle de *préfixe* de traces est celle de I-shuffle sur les mots. Soit $u \in \Sigma^*$ et $v \in \Sigma^\infty$. Le I-shuffle de u et v est l'ensemble des mots w tels que $\varphi(w) = \varphi(u)\varphi(v)$, et est noté $u \sqcup\!\sqcup_I v$. Ainsi, si l'on part d'une trace $t = rs$, avec $r \in \mathbf{M}(\Sigma, D)$ et $s \in \mathbf{R}(\Sigma, D)$, et si $w \in \varphi^{-1}(t)$, alors il existe des sous-mots u et v de w tels que $w \in u \sqcup\!\sqcup_I v$, avec $\varphi(u) = r$ et $\varphi(v) = s$.

2.1.5 Équivalence des représentations

Les notions de *traces* et de *graphes de dépendance* recouvrent en fait une même classe d'objets. Afin de mettre en évidence cette réalité, nous allons construire des fonctions $g : \mathbf{R}(\Sigma, D) \to \mathbf{G}(\Sigma, D)$ et $r : \mathbf{G}(\Sigma, D) \to \mathbf{R}(\Sigma, D)$ permettant de représenter respectivement une trace comme un graphe de dépendance et un graphe de dépendance comme une trace. La fonction g : $\mathbf{R}(\Sigma, D) \to \mathbf{G}(\Sigma, D)$ est définie comme suit : pour tout $[u] = [u_0u_1u_2\ldots]$ appartenant à $\mathbf{R}(\Sigma, D)$, $g([u]) = (E, \leq, \lambda)$ avec

- $E = \{\partial_{u_i}([u_0u_1\ldots u_i]), i \geq 0\}$.
- $\leq \subseteq E \times E$ est la relation de préfixe $\preceq$ sur les traces de E.
- $\forall i \geq 0$, $\lambda(\partial_{u_i}([u_0u_1\ldots u_i])) = u_i$.

$$
\begin{tikzcd}
{[a]} \arrow[dr] & & {[acba]} \arrow[r] & {[acbaa]} \arrow[dr] & \\
& {[acb]} \arrow[ur] \arrow[dr] & & & {[acbaccab]} \\
{[c]} \arrow[ur] & & {[acbc]} \arrow[r] & {[acbcc]} \arrow[ur] &
\end{tikzcd}
$$

FIG. 2.4 – *Diagramme de Hasse du graphe de dépendance $g([u])$, avant étiquetage par λ*

Cette définition ne dépend pas du représentant de la trace choisi car si $[u] = [u']$, alors à tout préfixe $u_0u_1 \dots u_i$ de u on fait correspondre l'unique préfixe $u'_0u'_1 \dots u'_k$ de u' tel que $u'_k = u_i$ et tel que le nombre d'occurrence de la lettre u_i soit le même dans $u_0 \dots u_i$ et dans $u'_0 \dots u'_k$. En appliquant le Lemme de Levi, on obtient $\partial_{u_i}([u_0 \dots u_i]) = \partial_{u'_k}([u'_0 \dots u'_k])$. Les mots u et u' génèrent donc le même ensemble d'événements E, et par conséquent le même graphe de dépendance F.

Exemple 4 *Pour l'alphabet de dépendance (Σ, D) avec $\Sigma = \{a, b, c\}$ et $D = \{(a, b); (b, a); (b, c); (c, b)\}$, on construit à partir de la trace $[u] = [acbaccab]$ le graphe de dépendance $g([u])$ de la Figure 2.4, que l'on a étiqueté volontairement par les préfixes $\partial_a([va])$ pour rendre la construction visible.*

Définissons à présent la fonction $r : \mathbf{G}(\Sigma, D) \to \mathbf{R}(\Sigma, D)$: pour tout graphe de dépendance $F = (E, \leq, \lambda)$, $\rho \in E^\infty$ est une *linéarisation* de F ssi tout $e \in E$ apparaît exactement une fois dans ρ et si, de plus, pour tous $e, e' \in E$ tels que $e \leq e'$, e apparaît avant e' dans ρ. Si $\rho = e_0e_1e_2 \dots$ est un mot de E^∞, $\lambda(\rho)$ désigne le mot $\lambda(e_0)\lambda(e_1)\lambda(e_2) \dots$. On définit alors $r(F)$ comme suit :

$$r(F) = [\lambda(\rho)] \text{ où } \rho \text{ est une linéarisation de } F.$$

Cette définition a un sens car si ρ et ρ' sont deux linéarisations de F, alors $\lambda(\rho) \sim_I \lambda(\rho')$. En effet, deux linéarisations d'un même graphe de dépendance F ne diffèrent que par l'ordonnancement d'événements qui ne sont pas comparables dans F, donc dont les étiquettes sont indépendantes. Deux événements ordonnés dans F le seront donc de la même manière dans toute linéarisation de F. Lorsqu'on projette deux linéarisations sur une D-clique, on obtient ainsi un unique mot, puisque tous les événements en jeu sont ordonnés dans F.

Ces fonctions de représentation permettent d'établir la proposition suivante :

Proposition 1 *Pour tout* $[u] \in \mathbf{R}(\Sigma, D)$*, et pour tout* $F \in \mathbf{G}(\Sigma, D)$*,*

$$r(g([u])) = [u]$$

$$g(r(F)) = F$$

Preuve. Soit $[u] = [u_0u_1u_2\ldots] \in \mathbf{R}(\Sigma, D)$. On construit à partir de $[u]$ le graphe de dépendance $g([u]) = F = (E, \leq, \lambda)$, où $E = \{\partial_a([va]) \mid [va] \preceq [u]\}$ et où $\leq = \preceq$ est la relation de préfixe sur ces traces. Soit $\rho = e_0e_1e_2\ldots$ tel que pour tout $i \geq 0$, $e_i = \partial_{u_i}([u_0u_1\ldots u_i]) \in E$. Alors ρ est une linéarisation évidente de F, et $\lambda(\rho) = u_i$, donc $r(g([u])) = [u]$.

Réciproquement, soit $F = (E, \leq, \lambda)$ un graphe de dépendance, et soit $\rho = e_0e_1e_2\ldots$ une linéarisation de ce graphe. Alors $r(F) = [\lambda(\rho)]$. Donc $g(r(F)) = (E', \leq', \lambda')$, avec $E' = \{\partial_{\lambda(e_i)}([\lambda(e_0\ldots e_i)]), i \geq 0\}$. Les graphes de dépendance F et F' sont alors isomorphes. □

La proposition 1 montre que l'on peut parler indifféremment de traces ou de graphes de dépendance. Désormais, on qualifiera de *trace* l'une ou l'autre de ces notions. En réalité, ces deux points de vue sont complémentaires, et permettent de se faire une idée à la fois plus complète et plus intéressante de la notion de trace : certaines propriétés sont plus faciles à établir lorsque l'on considère les traces comme des classes d'équivalence de mots, alors que d'autres sembleront plus naturelles lorsque l'on considère les traces comme des graphes.

2.2 Langages de traces

On a vu que l'ensemble $\mathbf{M}(\Sigma, D)$ des traces finies est un monoïde pour le produit de concaténation. Ce n'est pas le cas pour l'ensemble des traces réelles $\mathbf{R}(\Sigma, D)$. Nous commençons ici par préciser la notion de concaténation pour des traces réelles.

Pour toute trace réelle $F = (E, \leq, \lambda) \in \mathbf{R}(\Sigma, D)$, on définit l'alphabet à l'infini de F par

$$\mathrm{alphinf}(F) = \{a \in \Sigma \mid \lambda^{-1}(a) \text{ est infini}\}.$$

Il s'agit de l'ensemble des lettres de Σ ayant un nombre infini d'occurrences dans F.

Pour toutes traces $F_1 = (E_1, \leq_1, \lambda_1)$ et $F_2 = (E_2, \leq_2, \lambda_2)$ de $\mathbf{R}(\Sigma, D)$, si $\mathrm{alphinf}(F_1) \times \mathrm{alph}(F_2) \subseteq I$ alors on définit la concaténation de F_1 et F_2 comme $F_1.F_2 = F = (E, \leq, \lambda)$ avec :

- $E = E_1 \cup E_2$;
- $\leq = \leq_1 \cup \leq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 \mid (\lambda_1(e_1), \lambda_2(e_2)) \in D\}$;
- $\lambda = \lambda_1 \cup \lambda_2$.

Remarquons que si $F_1 \in \mathbf{M}(\Sigma, D)$ alors $\mathrm{alphinf}(F_1) = \emptyset$, donc la concaténation d'une trace finie et d'une trace réelle est toujours définie.

Si $(F_i)_{i \geq 0}$ est une suite infinie de traces réelles, avec $F_i = (E_i, \leq_i, \lambda_i)$, vérifiant $\forall j \geq 0$, $\forall i < j$, $\mathrm{alphinf}(F_i) \times \mathrm{alph}(F_j) \subseteq I$, alors on peut définir la ω-concaténation des $(F_i)_{i \geq 0}$ comme $F = (E, \leq, \lambda)$ où

- $E = \cup_{i \geq 0} E_i$;
- $\leq = (\cup_{i \geq 0} \leq_i) \cup \{(e, e') \in E_i \times E_j \mid i < j \text{ et } (\lambda_i(e), \lambda_j(e')) \in D\}$;
- $\lambda = \cup_{i \geq 0} \lambda_i$.

La condition $i < j \Rightarrow \mathrm{alphinf}(F_i) \times \mathrm{alph}(F_j) \subseteq I$ assure que tout événement de E a un passé fini. On note alors $F = \prod_{i \geq 0} F_i$. De plus, cette condition implique que seul un nombre fini des traces F_i peuvent être des traces infinies.

2.2.1 Langages rationnels

Si $R, S \subseteq \mathbf{R}(\Sigma, D)$ sont deux langages de traces réelles, leur concaténation est définie par

$$R.S = \{r.s \mid r \in R, s \in S \text{ et } \mathrm{alphinf}(r) \times \mathrm{alph}(s) \subseteq I\}.$$

On définit, pour $R \subseteq \mathbf{R}(\Sigma, D)$, $R^0 = \{[\varepsilon]\}$ et pour tout $n \geq 0$, $R^{n+1} = R^n.R$. Ceci permet de définir l'itération finie (ou $*$-itération) de R :

$$R^* = \cup_{n \geq 0} R^n.$$

L'itération infinie de R, (ou ω-itération) est définie comme l'ensemble des ω-concaténations de traces de R :

$$R^\omega = \{\prod_{i\geq 0} r_i \mid \forall i, r_i \in R \text{ et } \mathrm{alphinf}(r_i) \times \mathrm{alph}(r_j) \subseteq I \text{ pour tout } j > i\}$$

Définition 3 *La famille* Rat $\mathbf{R}(\Sigma, D)$ *des langages rationnels de traces réelles est la plus petite famille contenant l'ensemble vide et les singletons* $\{[a]\}$ *pour tout* $a \in \Sigma$, *et fermée pour les opérations d'union, de concaténation, et d'itérations finie et infinie sur les traces réelles.*

Le lien avec les langages de mots rationnels est établi par le théorème suivant :

Théorème 2 ([13, 15]) *Soit* $R \subseteq \mathbf{R}(\Sigma, D)$ *un langages de traces réelles. Les assertions suivantes sont équivalentes :*

(1) R *est rationnel.*

(2) Il existe un langage de mots rationnel $L \subseteq \mathrm{Rat}\ \Sigma^\infty$ *tel que* $R = \varphi(L)$, *où* φ *désigne l'application canonique de* Σ^∞ *dans* $\mathbf{R}(\Sigma, D)$.

2.2.2 Langages reconnaissables

La notion de *reconnaissabilité* est centrale dans la théorie des langages de mots, car elle permet de décrire les langages pouvant être reconnus (ou interprétés) par des mécanismes *finis*. De plus, elle permet d'établir un lien entre les langages dans leur généralité (ensembles quelconques de mots) et des outils algébriques (morphismes, monoïdes,...). L'intérêt de cette liaison est de pouvoir manipuler les langages (a priori objets infinis) à travers des objets finis. On peut généraliser cette notion de langage *reconnaissable* au cas des langages de traces.

L'ensemble $\mathbf{M}(\Sigma, D)$ des traces finies est un monoïde : il est clos par le produit de concaténation. D'une façon générale, si M et N sont deux monoïdes, une application $\eta : M \to N$ est un *morphisme* (de monoïdes) ssi pour tous $m, m' \in \mathbf{M}(\Sigma, D)$, η vérifie :

$$\eta(mn) = \eta(m)\eta(m')$$

et si de plus, en notant 1_M et 1_N les éléments neutres de M et N respectivement pour la concaténation, on a :

$$\eta(1_M) = 1_N.$$

Les définitions suivantes généralisent pour les traces les définitions classiques concernant les langages de mots.

Définition 4 *Un langage de traces finies $T \subseteq \mathbf{M}(\Sigma, D)$ est* reconnaissable *ssi il existe un monoïde fini M, et un morphisme de monoïdes $\eta : \mathbf{M}(\Sigma, D) \to M$ tels que*

$$T = \eta^{-1}(\eta(T))$$

Définition 5 *Un langage de traces réelles $T \subseteq \mathbf{R}(\Sigma, D)$ est* reconnaissable *ssi il existe un monoïde fini M, et un morphisme de monoïdes $\eta : \mathbf{M}(\Sigma, D) \to M$ tels que pour toute suite finie ou infinie (t_i) d'éléments de $\mathbf{M}(\Sigma, D)$ on ait :*

$$t_1 t_2 t_3 \ldots \in T \Rightarrow \eta^{-1}\eta(t_1)\eta^{-1}\eta(t_2)\eta^{-1}\eta(t_3)\ldots \subseteq T$$

On note Rec $\mathbf{R}(\Sigma, D)$ l'ensemble des langages reconnaissables de $\mathbf{R}(\Sigma, D)$. On peut alors énoncer le théorème suivant, qui relie les langages rationnels et reconnaissables :

Théorème 3 ([13]) Rec $\mathbf{R}(\Sigma, D) \subseteq$ Rat $\mathbf{R}(\Sigma, D)$.

Contrairement à ce qui se passe pour les mots, dans le cas des traces la réciproque est fausse : il existe des langages de traces rationnels qui ne sont pas reconnaissables. Par exemple, si les lettres a et b sont indépendantes, alors le langage de traces $T = \varphi(ab)^*$ est évidemment rationnel, mais il n'est pas reconnaissable, car $\varphi^{-1}(T)$ est l'ensemble des mots comportant autant de a que de b et le théorème suivant nous permet de conclure, en établissant le lien avec les langages de mots reconnaissables :

Théorème 4 ([13]) *Soit $T \subseteq \mathbf{R}(\Sigma, D)$ un langage de traces réelles. Les assertions suivantes sont équivalentes :*

(1) T est reconnaissable.

(2) $\varphi^{-1}(T) \subseteq \Sigma^\infty$ est un langage de mots reconnaissable.

Une sous-famille de langages de traces reconnaissables intéressante est celle des langages *sans-étoile*. Pour construire cette famille à partir des singletons, on n'autorise pas d'itération infinie comme pour les langages rationnels, mais on se permet d'utiliser la complémentation et l'intersection. Formellement, on définit les langages de traces sans-étoile comme suit :

Définition 6 *La famille* $\mathrm{SF}(\mathbf{M}(\Sigma, D))$ *des langages sans-étoile de traces finies est la plus petite famille contenant les singletons* $\{[a]\}$ *pour tout* $a \in \Sigma$ *fermée par les opérations booléennes (union, intersection, complémentation) et par la concaténation.*

Pour les langages de traces réelles, on a recours à la définition suivante :

Définition 7 *La famille* $\mathrm{SF}(\mathbf{R}(\Sigma, D))$ *des langages sans-étoile de traces réelles est la plus petite famille contenant* $\mathrm{SF}(\mathbf{M}(\Sigma, D))$ *fermée par la concaténation à gauche avec des langages de* $\mathrm{SF}(\mathbf{M}(\Sigma, D))$ *et par la complémentation dans* $\mathbf{R}(\Sigma, D)$.

Notons que les langages de $\mathrm{SF}(\mathbf{M}(\Sigma, D))$ étant des langages de traces finies, la concaténation à gauche avec ces langages est toujours définie dans $\mathbf{R}(\Sigma, D)$.

La famille $\mathrm{SF}(\mathbf{R}(\Sigma, D))$ est une sous-famille de Rec $\mathbf{R}(\Sigma, D)$, autrement dit tout langage sans-étoile est reconnaissable. En effet, tout singleton $\{[a]\}$ peut être reconnu par un morphisme qui d'une part distingue le a des autres lettres de Σ, et d'autre part distingue les traces d'une seule lettre des autres traces. De plus, on peut montrer que la famille des langages reconnaissables est fermée par les opérations d'union (par définition), d'intersection, et de complémentation.

En fait, on peut même caractériser les langages sans-étoile par le type des monoïdes dans lesquels ils sont reconnus. On dit qu'un monoïde M est *apériodique* ssi il existe un entier $k \geq 0$ tel que tout élément m de M vérifie l'identité $m^{k+1} = m^k$. Le théorème suivant caractérise alors les langages de traces *finies* sans-étoile :

Théorème 5 ([20]) *Un langage de traces* $L \subseteq \mathbf{M}(\Sigma, D)$ *est sans-étoile ssi il est reconnu dans un monoïde apériodique.*

2.2.3 Langages c-rationnels

On vient de voir que pour les traces réelles, les familles des langages reconnaissables et rationnels ne coïncident pas, contrairement à ce qui se passe pour les mots infinis : les langages de Σ^∞ reconnaissables et ω-rationnels forment une unique famille. Nous allons introduire une nouvelle famille de langages de traces réelles, appelés langages c-rationnels, afin de remédier à cette situation.

On dit qu'une trace réelle $F \in \mathbf{R}(\Sigma, D)$ est *connexe* si le graphe de dépendance F l'est. Si F n'est pas connexe, il existe une unique famille $\{F_1, F_2, \ldots, F_k\}$ de traces connexes non-vides deux-à-deux indépendantes telle que $F = F_1.F_2.\cdots.F_k$. Les traces $F_1, \ldots, F_k$ sont les *composantes connexes* de F. Pour toute trace réelle $F \in \mathbf{R}(\Sigma, D)$ on note $C(F)$ l'ensemble des composantes connexes de F. De plus, si $R \subseteq \mathbf{R}(\Sigma, D)$ est un langage de traces réelles, on note $C(R) = \cup_{F \in R} C(F)$. L'ensemble $C(R)$ est également un langage de traces réelles.

L'opérateur C sur les langages de traces réelles permet de définir deux opérations d'itérations nouvelles, appelées *itérations concurrentes*.

Soit $R \subseteq \mathbf{R}(\Sigma, D)$ un langage de traces réelles. Alors les itérations concurrentes finie et infini de R sont respectivement :

$$R^{c-*} = (C(T))^* \text{ et } R^{c-\omega} = (C(T))^\omega.$$

Définition 8 *La famille $c-\mathrm{Rat}\ \mathbf{R}(\Sigma, D)$ des langages c-rationnels de traces réelles est la plus petite famille contenant l'ensemble vide et les singletons $\{[a]\}$ qui soit fermée par les opérations d'union, de concaténation, et d'itérations concurrentes finie et infinie.*

Théorème 6 $c - \mathrm{Rat}\ \mathbf{R}(\Sigma, D) = \mathrm{Rec}\ \mathbf{R}(\Sigma, D)$.

Ce théorème [35, 14] généralise aux traces le théorème de Büchi sur les mots.

Le lecteur intéressé pourra consulter avec profit l'ouvrage "The Book of Traces" [9] pour des résultats plus détaillés et des généralisations diverses.

Chapitre 3

Traces et logiques

Dans ce chapitre nous commençons par un rappel concernant les logiques sur les mots. Nous rappelons l'existence de la logique du premier ordre et de la logique monadique du second ordre, puis nous exposons les principes d'une logique temporelle "canonique" : la logique LTL de Pnueli [39]. Ensuite, nous nous focalisons sur le cas des traces, en expliquant quelles sont les difficultés existantes dans ce cadre, et en présentant un bref panorama des logiques temporelles sur les traces. En particulier, nous explicitons la différence entre logiques *locales* et logiques *globales*.

3.1 Logiques sur les mots

3.1.1 Logiques FO et MSO($\leq$)

Nous nous intéressons en premier lieu à la logique monadique du second ordre ainsi qu'à une restriction syntaxique de cette logique : la logique du premier ordre. Dans ce cadre, un mot (infini) sur un alphabet Σ est considéré comme une application de l'ensemble $\mathbf{N}$ des entiers naturels dans Σ. Les formules de la logique monadique du second ordre (MSO($\leq$) pour *Monadic Second Order logic*) sont construites à l'aide de variables de positions notées $x, y, z, \ldots$ qui prennent leurs valeurs dans l'ensemble $\mathbf{N}$ (ces variables sont dites du premier ordre), de variables du second ordre notés $X, Y, Z, \ldots$ qui prennent leur valeur dans l'ensemble des parties de $\mathbf{N}$, des prédicats élémentaires suivants :

- $x < y$: la position de x est antérieure à celle de y;

- $R_a(x)$: la position de x à pour image a (i.e. est étiquetée par a);
- $X(y)$: la position de y appartient à l'ensemble de positions X;

et enfin des connecteurs booléens et des quantificateurs existentiels et universels sur les variables du premier et du second ordre.

A partir des connecteurs booléens, on peut construire les abréviations classiques du type $\Rightarrow$, $\Leftrightarrow$, etc... De même, on peut écrire des abréviations du type $X \subseteq Y$ pour la formule $\forall x(x \in X \Rightarrow x \in Y)$.

Une formule $\psi \in \mathrm{MSO}(\leq)$ appartient à la logique du premier ordre $\mathrm{FO}(\leq)$ (pour *First Order*) si elle ne contient aucune variable du second ordre. On dit qu'une variable x (resp. X) est *libre* dans une formule $\psi \in \mathrm{MSO}(\leq)$ si x n'est pas dans la portée d'un quantificateur portant sur x (resp. sur X).

Exemple 5 *Dans la formule $\psi = \exists x(y \leq x)$, y est une variable libre, alors que x ne l'est pas.*

On appelle *formule close* une formule qui ne comporte aucune variable libre. Toute variable y apparaissant est donc quantifiée. Une formule close ϕ est alors interprétée canoniquement sur un mot $w : \mathbf{N} \to \Sigma$ et on note $w \models \phi$ si l'*interprétation canonique* satisfait la formule ϕ. Cette interprétation est fondée sur la sémantique suivante : on appelle *évaluation* toute fonction E qui associe un entier à chaque variable du premier ordre et un sous-ensemble de $\mathbf{N}$ à toute variable du second ordre; alors

- $w \models_E R_a(x)$ ssi $\lambda(w_{E(x)}) = a$;
- $w \models_E x \leq y$ ssi $E(x) \leq E(y)$;
- $w \models_E \exists x \phi(x)$ ssi il existe une évaluation E' qui ne diffère de E qu'en x et telle que $w \models_{E'} \phi(x)$;
- etc...

et enfin, si ϕ est une formule *close*, alors

$$w \models \phi \text{ ssi } \exists E \mid w \models_E \phi.$$

Exemple 6 *Soit $\psi = \exists x(R_a(x))$. Les modèles de ψ sont les mots comportant au moins une occurrence de la lettre a.*

Soit $\psi' = \exists x(R_a(x) \wedge \forall y(y \leq x \Rightarrow R_a(y)))$. Les modèles de ψ' sont les mots dont la première lettre est a.

On dit qu'un langage de mots $L \subseteq \Sigma^\omega$ est définissable dans MSO($\leq$) (resp. dans FO($\leq$)) s'il existe une formule $\psi \in$ MSO($\leq$) (resp. $\psi \in$ FO($\leq$)) telle que

$$L = \{w \in \Sigma^\infty \mid w \models \psi\}.$$

Le pouvoir d'expression des logiques monadiques est caractérisé par les théorèmes suivants :

Théorème 7 (Büchi [3]) *Un langage de mots infinis est définissable dans la logique monadique du second ordre ssi il est régulier.*

Théorème 8 (McNaughton et Papert [30]) *Un langage de mots infinis est définissable dans la logique du premier ordre ssi il est sans-étoile.*

Notons que ces deux théorèmes restent valables dans le cadre des mots finis.

Les logiques FO($\leq$) et MSO($\leq$) ont donc un pouvoir d'expression parfaitement connu (et suffisant puisque MSO($\leq$) capture tous les langages reconnaissables). Néanmoins, la complexité de ces logiques (pour les problèmes de satisfaisabilité et de model-checking, cf. Chapitre 1) est non-élémentaire, même pour celle du premier ordre. Enfin, il est difficile de rendre compte de propriétés intéressantes (c'est-à-dire utiles pour la vérification) de façon concise dans de telles logiques (i.e. via des formules de taille réduite). Il est donc nécessaire de recourir à des logiques d'un autre type, qui permettent d'exprimer en peu de termes des propriétés intéressantes pour la vérification. Par ailleurs, dans le cadre de la vérification, il est plus naturel de se référer à l'agencement temporel des événements pour spécifier certains types de comportements. C'est pourquoi nous allons désormais nous préoccuper de logiques *temporelles*, en commençant par décrire une logique "canonique" sur les mots, la logique LTL de Pnueli [39].

3.1.2 Logiques temporelles

Une logique temporelle est un formalisme de description d'un modèle fondé sur la notion d'exécution. Une exécution est l'ensemble (ordonné ou partiellement ordonné) des états par lesquels passent un système au fil du temps. Une logique temporelle est donc constituée de propositions atomiques qui décrivent les propriétés des états rencontrés et d'opérateurs temporels qui décrivent les possibilités d'occurrences de ces propriétés au cours du temps.

Par exemple, un opérateur temporel pourra exprimer que la propriété qui le suit est toujours satisfaite, ou alors qu'elle le sera à un moment dans le futur. La puissance des logiques temporelles vient alors des possibilités d'imbrications de tels opérateurs. Nous allons commencer par le cas simple d'une logique temporelle sur les mots, la logique LTL de Pnueli [39], avant d'aborder un panorama des logiques temporelles existantes sur les traces.

La logique LTL sur les mots infinis On se donne un alphabet d'actions Σ fixé. La syntaxe de la logique LTL est la suivante :

$$\mathrm{LTL} ::= q_a \mid \alpha \vee \beta \mid \neg\alpha \mid X\alpha \mid \alpha\,\mathcal{U}\,\beta \text{ , où } a \in \Sigma.$$

Un modèle pour LTL est un mot $w \in \Sigma^\omega$. Les formules de LTL sont interprétées sur un couple (w, i), où i est un entier précisant la position courante dans le mot w. La sémantique d'une formule ϕ est alors définie inductivement comme suit :

- $w, i \models q_a$ ssi $w_i = a$, où w_i désigne la lettre d'indice i dans w.
- $w, i \models \alpha \vee \beta$ ssi $w, i \models \alpha$ ou $w, i \models \beta$.
- $w, i \models \neg\alpha$ ssi $w, i \not\models \alpha$.
- $w, i \models X\alpha$ ssi $w, i+1 \models \alpha$.
- $w, i \models \alpha\,\mathcal{U}\,\beta$ ssi il existe $j \geq i$ tel que $w, j \models \beta$ et pour tout k, si $i \leq k < j$ alors $w, k \models \alpha$.

On peut utiliser dans cette logique des abréviations classiques telles que :

$$\begin{aligned} \underline{tt} &\stackrel{\text{def}}{=} q_a \vee \neg q_a \text{ pour un } a \text{ fixé.} \\ \alpha \wedge \beta &\stackrel{\text{def}}{=} \neg(\neg\alpha \vee \neg\beta) \\ F\alpha &\stackrel{\text{def}}{=} \underline{tt}\,\mathcal{U}\,\alpha \\ G\alpha &\stackrel{\text{def}}{=} \neg F(\neg\alpha) \end{aligned}$$

Par exemple, la formule $\phi = G(Fq_a)$ signifie qu'à tout moment il existe un moment ultérieur où on lira la lettre a, i.e. que le mot w contient une infinité d'occurrences de a.

Une formule ϕ de LTL définit le langage suivant :

$$L(\phi) = \{w \in \Sigma^\omega \mid w, 0 \models \phi\},$$

l'indice 0 correspondant à la première lettre du mot considéré.

On voit facilement que la logique LTL peut s'exprimer dans la logique du premier ordre (sur les mots), car la définition des opérateurs temporels X et $\mathcal{U}$ ne fait appel qu'à des variables du premier ordre sur les positions. Par exemple, supposons que l'on connaisse les équivalents en logique du premier ordre des formules α et β de LTL, et notons respectivement α' et β' ces formules de FO($\leq$). Alors la formule $\phi = \alpha\, \mathcal{U}\, \beta$ de LTL possède un équivalent dans FO($\leq$) :

$$\phi' = \exists x(\beta'(x) \wedge \forall y(y < x \Rightarrow \alpha'(y))).$$

En fait, la réciproque est également vraie : toute formule de la logique du premier ordre (sur les mots) peut être exprimée par une formule équivalente de LTL.

Théorème 9 (Kamp [21, 4]) *Soit Σ un alphabet et $L \subseteq \Sigma^\omega$. Alors L est définissable dans LTL ssi L est définissable dans la logique du premier ordre.*

Par ailleurs, les problèmes de model-checking et de satisfaisabilité (cf. Chapitre 1) sont tous deux PSPACE-complet pour la logique LTL. Il s'agit d'une logique très utilisée en pratique, et pour laquelle de nombreux outils de vérification automatiques existent.

3.2 Logiques sur les traces

On s'intéresse maintenant au cas des logiques sur les traces. Avant d'aborder le problème existant des logiques temporelles, nous faisons un bref exposé des résultats disponibles concernant les logiques monadiques dans ce cadre.

3.2.1 Logiques FO et MSO($\leq$)

Les logiques FO($\leq$) et MSO($\leq$) peuvent être généralisées très facilement aux traces. En effet, il suffit pour cela d'interpréter les variables du premier ordre $x, y, z, \ldots$ comme représentant les sommets d'un graphe de dépendance $F = (E, \leq, \lambda)$ au lieu des positions dans un mot. De même, les variables du second ordre $X, Y, Z, \ldots$ représentent alors des ensembles de sommets du graphe F. La seule différence est qu'alors l'ordre sur les sommets n'est plus un ordre total mais un ordre partiel. L'interprétation des formules atomiques est identique au cas des mots, si ce n'est que le prédicat $x < y$ signifie x et y sont ordonnés *et* x est antérieure à y.

L'intérêt de cette généralisation est qu'elle permet d'obtenir des résultats d'expressivité analogues aux théorèmes de Büchi et de McNaughton-Papert sur les langages de mots :

Théorème 10 ([11, 10]) *Un langage de traces réelles est définissable dans* MSO($\leq$) *ssi il est reconnaissable.*

Notons qu'on parle ici de langages de traces réelles *reconnaissables* et non *rationnels*, puisque ces deux notions diffèrent pour les traces réelles. On rappelle néanmoins que les langages de traces réelles reconnaissables sont exactement les langages c-rationnels (cf. Chapitre 2).

Pour la logique du premier ordre, on a la caractérisation suivante :

Théorème 11 ([11, 10]) *Un langage de traces réelles est définissable dans* FO($\leq$) *ssi il est sans-étoile.*

Dans le cas des mots, le fait que l'usage de la logique LTL soit largement répandu fait de FO($\leq$) (grâce au théorème de Kamp) une *référence* en terme d'expressivité. C'est pourquoi, dans le cas des traces, on cherchera souvent à comparer le pouvoir d'expression d'une logique à celui de la logique du premier ordre FO($\leq$). En particulier, on dit d'une logique qu'elle est *expressivement complète* si elle a le même pouvoir d'expression que FO($\leq$).

3.2.2 Logiques temporelles

Lorsqu'on veut développer des outils de vérification dans le cadre des systèmes concurrents (et en particulier dans le cadre des traces), une première approche consiste à considérer des logiques linéaires, i.e. des logiques sur les mots représentant les linéarisations des exécutions concurrentes. En particulier, on est tenté d'utiliser la logique LTL pour spécifier et vérifier des systèmes concurrents. En pratique, cette approche est entravée par deux facteurs :

- le problème de l'explosion combinatoire des états (problème inhérent aux systèmes concurrents);

- le fait que toutes les formules de LTL ne sont pas *robustes*.

On rappelle que, dans le cadre des traces, où les actions d'une exécution ne sont pas que partiellement ordonnées, une formule est dite *robuste* si lorsqu'elle est satisfaite par une linéarisation d'une trace, alors elle est satisfaite par toutes les linéarisations de cette même trace. En particulier, les formules de LTL faisant intervenir l'opérateur *next* (X) ne sont a priori pas robustes : par exemple, la formule $q_a \wedge X q_b$ est satisfaite par les mots commençant par ab, mais pas par ceux commençant par ba, et n'est donc pas robuste si les actions a et b sont indépendantes. Par ailleurs, le nombre de linéarisations équivalentes croît exponentiellement avec le nombre de processus concurrents dans le système considéré. Une solution (partielle) à ce problème consiste à développer des techniques qui, utilisées sur des propriétés *robustes*, permettent de n'avoir à vérifier le model-checking que sur un seul représentant par classe d'équivalence des linéarisations (une classe est un ensemble de linéarisations d'une même exécution concurrente). Cette approche a été notamment étudié par Godefroid et Wolper d'une part [17, 18, 19] et par Peled d'autre part [38]. Ces techniques sont dites de *réduction d'ordre partiel*.

Afin d'éviter les difficultés dûes à la nature non-robuste de certaines propriétés de logiques comme LTL, et afin d'attaquer en profondeur le problème de l'explosion combinatoire, on est naturellement amené à s'intéresser à des logiques temporelles dont les modèles d'interprétation sont les traces de Mazurkiewicz, en raison de leur nature intrinsèquement distribuée. L'étude de telles logiques est relativement récent, les premières logiques temporelles sur les traces ayant été proposées en 1994. Nous dressons à présent un bref panorama de ces logiques, en expliquant en quoi notre travail a permis de faire avancer la compréhension et la mise en pratique de ces logiques.

Logiques locales vs. logiques globales

A l'intérieur du domaine des logiques temporelles sur les traces, deux approches distinctes sont possibles. Il s'agit en fait de deux classes de généralisations des logiques linéaires. Afin de préciser la différence entre ces deux sortes de généralisations, rappelons q'un alphabet de dépendance (Σ, D) peut toujours être vu comme résultant d'un alphabet distribué, i.e de la mise en parallèle d'un nombre fini de processus (cf. Chapitre 2). Chacun de ces processus est par nature linéaire, c'est-à-dire que les actions qui s'y déroulent sont totalement ordonnées. Ce n'est qu'en mettant en concurrence ces processus que l'on obtient un ordre partiel sur l'ensemble des actions.

Les logiques temporelles *locales* sur les traces sont obtenues en affectant à chaque processus une logique linéaire, ce qui permet de décrire localement les propriétés des exécutions de *chaque* processus, et en combinant les opérateurs issus de chacune de ces logiques. Par combinaisons, nous entendons ici nous seulement les combinaisons booléennes mais aussi les imbrications d'opérateurs relatifs à des processus distincts.

Au contraire, dans une logique temporelle *globale* sur les traces, les opérateurs sont définis sur les états globaux du système, et non sur les états relatifs à un processus particulier. Par exemple, un opérateur *next* global spécifiera l'avancement du système via n'importe lequel de ces processus linéaires sous-jacents, et non l'avancement de tel processus en particulier. Bien que dans le cas d'un opérateur *next* les sémantiques locales et globales soient relativement proches, nous verrons que ce n'est pas le cas pour des opérateurs plus complexe, et en particulier qu'il semble souvent difficile (ou impossible) de trouver un équivalent exprimé localement de certains opérateurs globaux.

La distinction logiques locales / logiques globales étant faite, nous établissons maintenant un panorama des logiques temporelles existantes pour les traces.

Un panorama des logiques sur les traces

La logique TrPTL La première logique temporelle interprétée directement sur des traces étant apparu est la logique TrPTL (*Trace Propositional linear Time Logic*) de Thiagarajan [44]. La syntaxe de cette logique s'inspire de la logique LTL de Pnueli sur les mots [39], mais il s'agit d'une généralisation *locale* de cette logique : des opérateurs temporels de type *next* et *until* sont affectés à chacun des processus composant l'alphabet distribué. La syntaxe de cette logique est la suivante :

$$\mathrm{TrPTL} ::= q@p \mid \neg\alpha \mid \alpha \vee \beta \mid @_p\alpha \mid \alpha\, \mathcal{U}_p\, \beta,$$

où p appartient à l'ensemble P des processus, a appartient à l'alphabet Σ_p de ce processus, et q est une proposition atomique dont l'ensemble est noté AP.

Les formules de TrPTL sont interprétées sur des modèles de la forme (F, c), où $F = (E, \leq, \lambda)$ est une trace réelle sur l'alphabet distribué considéré et c est une configuration de la trace F. Pour un processus p fixé, les opérateurs temporels $@_p$ (*next* indexé par une action) et $\mathcal{U}_p$ (*until*) ont une sémantique proche des opérateurs X et $\mathcal{U}$ de LTL, mais ne concernent que les configurations relatives au processus p (i.e. les p-vues, cf Chapitre 2). Cette logique sera étudiée en détail au Chapitre 4, où nous montrerons un résultat de décomposition des formules proche du Théorème de Mezei sur les mots [2].

On ne connaît pas l'expressivité exacte de cette logique, mais on sait qu'elle est *au plus* aussi expressive que FO($\leq$). En ce qui concerne sa complexité, les problèmes de satisfaisabilité et du model-checking sont $PSPACE$-complets pour TrPTL. En particulier, il est possible de transformer toute formule de TrPTL en un automate reconnaissant le même langage; cette traduction s'effectue vers des automates particuliers, dits *asynchrones*, qui possèdent par nature un aspect distribué : un automate *asynchrone* est une famille finie d'automates *séquentiels* qui possèdent leurs propres alphabets d'actions, et qui se synchronisent sur certaines actions communes.

De plus, Mukund et Thiagarajan ont étudié également la possibilité d'une traduction de complexité plus faible des formules vers des automates asynchrones pour des fragments de la logique TrPTL, nommées TrPTL$^{\otimes}$ et TrPTL^{con}, définis via des restrictions syntaxiques [33]. En outre, Thiagarajan a montré que le fragment TrPTL$^{\otimes}$ peut être mis en relation de façon simple avec un sous-ensemble de formules de la logique LTL sur les mots noté LTL$^{\otimes}$, défini syntaxiquement, et dont toutes les formules sont robustes [45]. Néanmoins, les restrictions syntaxiques imposées sur TrPTL$^{\otimes}$ sont très contraignantes, et le fragment correspondant LTL$^{\otimes}$ ne contient pas toutes les formules robustes de LTL. Enfin, on sait que la logique TrPTL$^{\otimes}$ est strictement moins expressive que TrPTL.

D'autres classes de formules, intermédiaires entre TrPTL$^{\otimes}$ et TrPTL ont aussi été étudiés par Ramanujam [40], Parce qu'il semblait difficile de caractériser le pouvoir d'expression de ces logiques *locales*, les recherches dans ce domaine se sont alors principalement focalisées sur les logiques *globales*.

La logique TLPO Une logique temporelle *globale* sur les traces, nommée TLPO a été introduite par Ebinger en 1994 [10]. Les opérateurs temporels de cette logique sont du type X (*next*), $\mathcal{U}$ (*until*), et leurs symmétriques Y (*previous*) et $\mathcal{S}$ (*since*).

Ebinger a montré que cette logique était *expressivement complète*, i.e. avait le même pouvoir d'expression que FO($\leq$), mais cette preuve nécessite l'utilisation de tous les opérateurs, aussi bien futurs que passés. On ne sait pas si le fragment de TLPO ne contenant que des opérateurs de futur est encore expressivement complet. Cette situation est à comparer avec celle de la logique LTL sur les mots, pour laquelle on sait que l'ajout d'opérateurs passés (i.e. *previous* et *since*) n'augmentent pas le pouvoir d'expression [52].

La logique TLPO a été relativement peu étudiée, si ce n'est par Kuske, qui a proposé une extension de cette logique grâce à des quantificateurs, ce qui permet d'obtenir un pouvoir d'expression équivalent à celui de MSO($\leq$), et par conséquent de décrire tous les langages de traces reconnaissables [27]. De plus, cette généralisation ne nécessite plus d'opérateurs passés, contrairement à TLPO.

La complexité de TLPO n'est pas connue, mais il semble que pour les traces la coexistence d'opérateurs futurs et passés rende difficile toute traduction des formules vers des automates. L'intérêt de cette logique est donc plutôt d'ordre historique, car il s'agit de la première logique expressivement complète sur les traces, mais son utilité pratique semble faible.

La logique LTrL Afin de remédier aux problèmes d'expressivité de la logique TrPTL, Thiagarajan et Walukiewicz ont proposé une autre logique temporelle sur les traces dénommée LTrL (*Linear time Trace Logic*). Cette logique est expressivement complète [46]. Il s'agit d'une généralisation *globale* de la logique LTL sur les mots. La syntaxe de LTrL est la suivante :

$$\text{LTrL} ::= \underline{tt} \mid \neg\alpha \mid \alpha \vee \beta \mid ⓐ\alpha \mid \alpha\, \mathcal{U}\, \beta \mid \underline{a},$$

où a appartient à l'alphabet Σ considéré. L'opérateur ⓐ est un *next global* indexé par l'action a, i.e. permettant d'avancer d'une configuration en effectuant l'action a, tandis que l'opérateur $\mathcal{U}$ est un *until global*, dont la sémantique est la même que celle du *until* de LTL, en remplaçant les préfixes de mots par les préfixes de traces. Enfin, les constantes de type $\underline{a}$ permettent de parler du *passé immédiat* d'une configuration, c'est-à-dire d'en décrire les éléments maximaux, et sont appelées des formules de présent. Cette logique est étudiée en détail au Chapitre 5, dans lequel nous proposons un algorithme de transformation modulaire des formules de LTrL en automates de Büchi reconnaissant le même langage. Nous verrons en particulier que la difficulté majeure de cette transformation apparaît lors de l'utilisation de l'opérateur $\mathcal{U}$.

Alors que le problème de la satisfaisabilité d'une formule est PSPACE-complet pour LTL[43], Walukiewicz a montré que ce problème devenait non-élémentaire pour LTrL [50]. Un tel saut de complexité paraît étonnant, et est dû à l'opérateur global $\mathcal{U}$. En effet, en remplaçant cet opérateur binaire par un opérateur unaire de type F (*eventually*) la satisfaisabilité devient un problème EXPSPACE [50], mais la complétude est perdue (i.e. la logique n'est plus aussi expressive que FO($\leq$)). Un inconvénient mineur de la logique LTrL est la présence des formules de présent, qui font appel, pour leur sémantique, à des propriétés de la configuration actuelle.

En particulier, ces formules sont utilisées de façon intensive dans la preuve de la complétude de LTrL. On ne sait pas si le fragment de LTrL ne contenant pas les formules de présent est encore expressivement complet.

Dans le Chapitre 6, nous montrons comment on peut utiliser cette logique pour spécifier des réseaux de signaux de transitions (formalisme utilisé dans la communauté *architecture* permettant de décrire des réseaux de composants electroniques), et nous prouvons un algorithme efficace (polynômial) de model-checking dans ce contexte, ceci malgré le caractère non-élémentaire de ce problème pour la logique LTrL dans le cadre général des traces.

La logique LTL$_f$ Plus récemment, Diekert et Gastin ont proposé une logique temporelle sur les traces notée LTL$_f$, qui diffère de LTrL en ceci qu'elle ne possède pas de formules de présent [7]. Tous les opérateurs temporels de cette logique sont donc de type *pur futur*. Afin d'obtenir la complétude, Diekert et Gastin ont recours, en plus des opérateurs *next* et *until*, à des opérateurs de type $\langle B^*\rangle\alpha$, qui exprime que α sera satisfaite dans une configuration future à laquelle on peut parvenir en n'effectuant que des actions de l'ensemble $B \subseteq \Sigma$. Ces nouveaux opérateurs agissent donc comme des filtres, qui ne laissent passer que les actions de B. Notons que dans le cas d'un alphabet où toutes les lettres sont dépendantes, on retrouve alors exactement la sémantique de la logique LTL, et qu'alors les filtres $\langle B^*\rangle$ peuvent s'exprimer via l'opérateur $\mathcal{U}$. Dans le cas général, on ne sait pas si cela est encore possible. En outre, il n'existe pas de traductions simples entre les logiques LTrL et LTL$_f$, pourtant proches.

La logique LTL$_f$ est expressivement complète, mais sa complexité est, comme celle de LTrL, non-élémentaire. Son principal intérêt (théorique) réside donc dans l'absence de formules de présent. Sur le plan pratique, on peut penser que les filtres $\langle B^*\rangle$ sont des outils intéressants de spécification. De plus, il est aisé d'adapter les constructions modulaires du Chapitre 5 au cas de ces opérateurs, ce qui permet d'obtenir une procédure de décision modulaire pour LTL$_f$. Ce point particulier sera évoqué plus en détail en conclusion du Chapitre 5.

3.3 Conclusion

De ce panorama des principales logiques temporelles sur les traces, on peut tirer quelques observations. Une “bonne” logique temporelle sur les traces devrait satisfaire aux exigences suivantes :

- être suffisamment expressive (si possible équivalente à FO($\leq$));
- posséder une complexité raisonnable pour les problèmes de model-checking et de satisfaisabilité;
- permettre d'exprimer des propriétés utiles à la vérification par des formules de petites tailles.

Il semble que le caractère local d'une logique sur les traces favorise une complexité faible, alors qu'un caractère global favoriserait une plus grande expressivité. La recherche d'une logique temporelle “idéale” pour les traces, c'est-à-dire satisfaisant aux critères énonçés plus haut, s'apparente donc à celle d'un compromis entre localité et globalité, ou entre complexité et expressivité.

Il existe déjà de nombreuses logiques temporelles pour les traces, des logiques *locales* comme des logiques *globales*, mais aucune ne satisfait pour l'instant à l'ensemble des critères requis pour être un outil de vérification réellement efficace. En fait, l'existence d'une telle logique, i.e. une logique temporelle élémentaire et expressivement complète, demeure pour l'instant un problème ouvert. A défaut d'exhiber une logique temporelle idéale, notre travail a permis de mieux comprendre les utilisations possibles des logiques existantes (notamment TrPTL et LTrL) et a fourni des résultats à la fois théoriques et pratiques dans le cadre des logiques temporelles sur les traces de Mazurkiewicz.

Chapitre 4

Un théorème de décomposition

Dans ce chapitre, on présente de façon détaillée la logique temporelle sur les traces TrPTL de Thiagarajan [44]. Il s'agit d'une logique temporelle linéaire, dont la sémantique des opérateurs s'inpire de la logique sur les mots LTL de Pnueli [39] (cf. Chapitre 3), et qui en est une généralisation dite "locale", en ceci que tout opérateur temporel est lié à un processus particulier, donc à des états locaux, et non aux états globaux du système.

Après avoir décrit le cadre de cette logique, sa syntaxe et sa sémantique, nous nous intéresserons aux problèmes d'expressivité et de complexité qu'elle engendre. On rappellera brièvement les résultats de Mukund et Thiagarajan concernant ces différents points, et on montrera comment on peut exprimer dans cette logique des propriétés temporelles "globales" (dans une certaine mesure), et ce malgré le caractère local des opérateurs de TrPTL.

Le résultat principal que nous établissons dans ce chapitre est un théorème de décomposition : dans le contexte particulier d'un alphabet de dépendance non-connexe, nous montrons que toute formule de TrPTL peut être décomposée, de façon effective, en une combinaison booléenne de formules dont l'interprétation ne dépend que d'une composante connexe de (Σ, D) à la fois. Ce théorème est à rapprocher du théorème de Mezei [2], qui énonce que tout langage reconnaissable dans un produit $M_1 \times M_2$ de monoïdes est une union finie de produits du type $L_1 \times L_2$, où L_i est un langage reconnaissable de M_i pour $i \in \{1, 2\}$.

Nous montrerons enfin en quoi un tel théorème de décomposition est utile dans le cadre de la vérification. Ce théorème et ses conséquences ont fait l'objet d'une publication lors de la conférence MFCS'97 à Bratislava, Slovaquie [31].

4.1 Le cadre de la logique TrPTL

Le point de départ de la logique temporelle TrPTL est un alphabet distribué $\{\Sigma_p\}_{p\in\mathcal{P}}$, l'ensemble des processus $\mathcal{P}$ étant fini, de cardinal k (cf. Chapitre 2). On rappelle que Σ_p désigne l'alphabet des actions impliquant le processus p. Soit (Σ, D) l'alphabet de dépendance induit par $\{\Sigma_p\}_{p\in\mathcal{P}}$. L'alphabet $\Sigma = \cup_{p\in\mathcal{P}}\Sigma_p$ comprend toutes les actions possibles du système de processus $\mathcal{P}$. On rappelle également qu'à toute action $a \in \Sigma$ on associe l'ensemble des processus $pr(a) = \{p \in \mathcal{P} \mid a \in \Sigma_p\}$ qui participe à l'action synchronisée a, et que la relation de dépendance D est définie par : $(a, b) \in D$ ssi $pr(a) \cap pr(b) \neq \emptyset$.

Par ailleurs, la logique TrPTL fait intervenir des propositions atomiques, dont l'ensemble, au plus dénombrable, est noté $AP = \{q, r, s, \ldots\}$. Une des particularité de la logique TrPTL est de ne faire usage de ces propositions atomiques que de façon locale : une proposition atomique n'est jamais évaluée dans un état global du système.

Enfin, TrPTL fait appel à la notion de *vue locale* d'une configuration par un processus. Pour cela, on s'intéresse, dans une trace donnée $F = (E, \leq, \lambda)$ aux sommets (ou événements) dont l'étiquette est dans Σ_p, pour un processus $p \in \mathcal{P}$ fixé. On appelle *p-événements* de F l'ensemble E_p suivant :

$$E_p = \{e \in E \mid \lambda(e) \in \Sigma_p\}$$

On peut alors définir aisément la notion de *vue locale* :

Définition 9 (Vue locale d'une configuration) *Soit $F = (E, \leq, \lambda)$ une trace sur (Σ, D), et soit $p \in \mathcal{P}$ un processus. Pour toute configuration $c \in C_F$, la p-*vue *de c, notée $\downarrow_p c$ est l'ensemble des événements de E dominés par un événement de c d'étiquette dans Σ_p :*

$$\begin{aligned} \downarrow_p c &= \downarrow(c \cap E_p) \\ &= \{e \in E \mid \exists e' \in (c \cap E_p), e \leq e'\}. \end{aligned}$$

Autrement dit, on a

$$\downarrow_p c = \partial_{\Sigma_p}(c)$$

en utilisant les notations du Chapitre 2.

Remarque 5 *Pour toute trace F et toute configuration $c \in C_F$, la p-vue de c est toujours une configuration, i.e. un ensemble fini de sommets clos par le passé.*

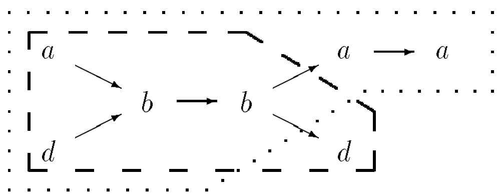

FIG. 4.1 – *Une configuration c et ses vues partielles $\downarrow_1(c)$ et $\downarrow_2(c)$.*

Exemple 7 *Soit (Σ, D) l'alphabet de dépendance engendré par l'alphabet distribué $\{\Sigma_1, \Sigma_2\}$, où $\Sigma_1 = \{a, b\}$ et $\Sigma_2 = \{b, d\}$. Seules les actions a et d sont donc indépendantes. Considérons la configuration c de la Figure 4.1. La vue $\downarrow_1(c)$ est entourée avec des points, alors que la vue $\downarrow_2(c)$ est entourée avec des traits gras. On remarque que $\downarrow_2(c)$ ne comprend pas les deux dernières actions a, puisqu'elles ne font appel qu'au seul processus 1. Le processus 2 ne peut donc pas à ce moment être "au courant" de ces actions.*

4.2 Syntaxe et sémantique de TrPTL

La syntaxe de la logique TrPTL est calquée sur celle de LTL(cf. Chapitre 3), si ce n'est que les opérateurs temporels, de même que les propositions atomiques, sont indexés par les processus.

4.2.1 Aspect général de TrPTL

L'ensemble Φ des formules de TrPTL peut être décrit inductivement de la façon suivante :

$$\Phi ::= q@p \mid \neg\alpha \mid \alpha \vee \beta \mid @_p\alpha \mid \alpha\ \mathcal{U}_p\ \beta \text{ avec } p \in \mathcal{P}, a \in \Sigma \text{ et } q \in AP.$$

Pour interpréter une formule de TrPTL, on a besoin non seulement d'une trace F mais également d'une *fonction d'évaluation* $V : C_F \to (2^{AP})^k$, où l'on rappelle que k est le cardinal de l'ensemble des processus $\mathcal{P}$. Cette fonction associe à toute configuration $c \in C_F$ un vecteur de dimension k, dont la p-ème composante, notée $V(c)[p]$ est (intuitivement) le sous-ensemble des propositions atomiques qui sont vérifiées par le processus p dans la configuration c.

Afin de conserver l'aspect distribué du modèle, la fonction d'évaluation V doit de plus satisfaire la propriété de localité suivante :

$$\forall c, c' \in C_F, c \xrightarrow{a} c' \Rightarrow \forall p \notin pr(a), V(c)[p] = V(c')[p]$$

Autrement dit, lorsqu'on passe d'une configuration c à une configuration c' en effectuant une action a, les propositions atomiques vérifiées par un processus qui n'est pas impliqué dans a ne doivent pas varier. Ceci traduit le fait qu'un tel processus ne voit pas sa vue locale changer, comme le montre la proposition suivante :

Proposition 2 *Soient $c, c' \in C_F$ tels que $c \xrightarrow{a} c'$, et soit un processus p n'appartenant pas à $pr(a)$, i.e. $a \notin \Sigma_p$. Alors on a l'égalité suivante :*

$$\downarrow_p c = \downarrow_p c'$$

Preuve. Puisque $c \xrightarrow{a} c'$, il existe un événement $e \in E$ d'étiquette a, tel que $c' = c \cup \{e\}$. Puisque $a \notin \Sigma_p$, l'événement e n'est pas dans E_p, donc $E_p \cap c = E_p \cap c'$, ce qui conduit à l'égalité des p-vues : $\downarrow_p c = \downarrow_p c'$. □

Remarque 6 *La Proposition 2 montre clairement qu'une autre alternative aurait été de définir la fonction d'évaluation comme un ensemble de k applications $(V_p)_{p \in \mathcal{P}}$, l'application V_p allant de l'ensemble des p-vues de la trace F dans l'ensemble des parties de AP. Cette définition alternative aurait permis de se passer de la condition de localité exprimée plus haut.*

4.2.2 Modèles et interprétation

Définition 10 *Un* modèle *pour la logique temporelle TrPTL est un couple $M = (F, V)$ où $F = (E, \leq, \lambda)$ est une trace sur l'alphabet de dépendance (Σ, D), et V une fonction d'évaluation satisfaisant la condition de localité exprimée plus haut.*

Une formule de TrPTL s'interprète dans un modèle $M = (F, V)$, et pour une configuration donnée c de F. La sémantique des formules de TrPTL est définie inductivement sur leur structure, de la façon suivante :

- $c \models_M q@p$ ssi $q \in V(c)[p]$.

- $c \models_M \neg\alpha$ ssi $c \not\models_M \alpha$;
- $c \models_M \alpha \vee \beta$ ssi $c \models_M \alpha$ ou $c \models_M \beta$;
- $c \models_M @_p\alpha$ ssi $a \in \Sigma_p$ et $\exists c' \in C_F$ tels que $\downarrow_p c \xrightarrow{a} c'$ et $c' \models_M \alpha$;
- $c \models_M \alpha\, \mathcal{U}_p\, \beta$ ssi $\exists c' \in C_F \mid c \subseteq c'$ vérifiant $\downarrow_p c' \models_M \beta$, et $\forall c'' \in C_F$, si $\downarrow_p c \subseteq \downarrow_p c'' \subset \downarrow_p c$ alors $\downarrow_p c'' \models_M \alpha$.

Les opérateurs booléens $\neg$ et $\vee$ ont donc leur sémantique habituelle. Les propositions atomiques du type $p@i$, qui se lit "p en i", sont évalués pour un processus donné. L'opérateur $@_p$, exprime le fait que la formule qui le suit devra être satisfaite lors de la p-vue suivante, que l'on devra atteindre en effectuant une action a. Enfin, une formule du type $\alpha\, \mathcal{U}_p\, \beta$ est satisfaite en c, s'il existe une future p-vue en laquelle β est satisfaite, et si α est satisfaite en toutes les p-vues intermédiaires.

Par ailleurs, on utilisera par la suite les abréviations courantes du calcul booléen, comme par exemple $\alpha \wedge \beta$ pour la formule $\neg(\neg\alpha \vee \neg\beta)$. De plus, on définit la formule constante $\underline{tt}$ par

$$\underline{tt} \stackrel{\text{def}}{=} q@1 \vee \neg q@1$$

où q est une proposition atomique quelconque fixée. La formule $\underline{tt}$ est ainsi vérifiée en toute configuration.

Exemple 8 *Considérons l'alphabet distribué $\{\Sigma_1, \Sigma_2\}$, avec $\Sigma_1 = \{a, b\}$ et $\Sigma_2 = \{b, d\}$. Alors la formule*

$$\phi = (p@1)\, \mathcal{U}_1\, (q@1)$$

spécifie que le processus 1 doit (localement) satisfaire la proposition atomique p, jusqu'à un certain moment où il doit satisfaire q. La formule

$$\psi = (p@2)\, \mathcal{U}_1\, (q@2)$$

spécifie que les 2-vues des 1-vues doivent satisfaire p jusqu'à un moment où q est satisfaite. De telles configurations sont alors à la fois des 1-vues et des 2-vues. Comme le seul élément commun à Σ_1 et Σ_2 est $\{b\}$, la formule ψ spécifie un "p until *q" sur les configurations où b est le seul élément maximal, i.e. sur les configurations de "synchronisation" (au sens où les deux composantes ont les mêmes vues de ces configurations).*

Remarque 7 *Dans le cas particulier où l'on a un unique processus, on retrouve la sémantique classique de la logique LTL.*

Définition 11 (Satisfaisabilité) *On dit qu'une formule α de TrPTL est* satisfaisable *ssi il existe un modèle $M = (F, V)$ et une configuration $c \in C_F$ tels que :*

$$c \models_M \alpha$$

Définition 12 (Satisfaisabilité initiale) *On dit qu'une formule α de TrPTL est* initialement satisfaisable *ssi il existe un modèle $M = (F, V)$ tel que :*

$$\emptyset \models_M \alpha$$

4.2.3 Une interprétation alternative

La présentation qui vient d'être faite de la sémantique de TrPTL est celle de l'article original de Thiagarajan. Afin de mieux comprendre le caractère local de cette logique temporelle, nous allons exprimer cette sémantique de façon quelque peu différente.

Pour toute configuration c d'une trace F, et tout processus $p \in \mathcal{P}$, la p-vue de c est définie par $\downarrow(c \cap E_p)$. Or les événements de E_p sont ceux de E dont l'étiquette est dans Σ_p; ces événements sont donc totalement ordonnés. On note désormais C_F^p l'ensemble des p-vues de F, i.e. l'ensemble

$$C_F^p = \{\downarrow_p c \mid c \in C_F\}.$$

On a évidemment $C_F^p \subseteq C_F$, puisque toute p-vue est une configuration. De plus, une p-vue d'une configuration c possède toujours un élément maximal unique : c'est le plus grand événement de $c \cap E_p$. Son unicité provient du fait que les événements de E_p sont totalement ordonnés. On notera donc $\lambda_p(c)$ l'étiquette de l'événement maximal de $\downarrow_p c$ pour toute configuration $c \in C_F$.

Si deux configurations c et c' sont dans C_F^p, alors elles sont nécessairement ordonnées par inclusion : on a soit $c \subseteq c'$, soit $c' \subseteq c$. Ceci provient du fait que l'ensemble E_p est totalement ordonné.

Définition 13 *Puisque C_F^p est totalement ordonné pour l'inclusion, pour toute p-vue $c \in C_F^i$, on note $s_p(c)$ le successeur immédiat de c dans C_F^p, s'il existe, sinon on pose $s_p(c) = \emptyset$, et on note $S_p(c) = \{c' \in C_F^p \mid c \subseteq c'\}$ l'ensemble des successeurs de c dans C_F^p.*

Pour un modèle $M = (F, V)$ de TrPTL, on peut à présent donner une nouvelle interprétation des formules de la logique temporelle, que l'on notera $\models'_M$ afin de la différencier de l'interprétation précédente. Cette "nouvelle" sémantique est définie inductivement comme suit :

- $c \models'_M q@p$ ssi $q \in V(c)[p]$.
- $c \models'_M \neg\alpha$ ssi $c \not\models'_M \alpha$;
- $c \models'_M \alpha \vee \beta$ ssi $c \models'_M \alpha$ ou $c \models'_M \beta$;
- $c \models'_M @_p\alpha$ ssi $\lambda_p(s_p(\downarrow_p c)) = a$, et $s_p(\downarrow_p c) \models'_M \alpha$;
- $c \models'_M \alpha\ \mathcal{U}_p\ \beta$ ssi il existe $c' \in S_p(\downarrow_p c)$ vérifiant $c' \models'_M \beta$, et pour tout $c'' \in C_F^p$, si $\downarrow_p c \subseteq c'' \subset c'$ alors $c'' \models'_M \alpha$.

La proposition suivante montre l'équivalence de ces deux interprétations :

Proposition 3 *Soit ϕ une formule de TrPTL pour l'alphabet de dépendance (Σ, D), et soit $M = (F, V)$ un modèle pour cet alphabet. Pour toute configuration $c \in C_F$, on a :*

$$c \models_M \phi \Leftrightarrow c \models'_M \phi$$

Preuve. On montre l'équivalence proposée par induction sur la structure de la formule ϕ. Différents cas se présentent :

- Si $\phi = q@p$, alors trivialement $c \models_M \phi$ ssi $c \models'_M \phi$.
- Si $\phi = \neg\alpha$, on applique l'hypothèse d'induction à la formule α.
- Si $\phi = \alpha \vee \beta$, on applique l'hypothèse d'induction aux formules α et β.
- Si $\phi = @_p\alpha$, alors $c \models_M \phi$ implique qu'il existe $c' \in C_F$ telle que $\downarrow_p c \xrightarrow{a} c'$ et $c' \models_M \alpha$. Alors c' est bien une p-vue et même $c' = s_p(\downarrow_p c)$, ce qui permet de conclure par induction sur α. Réciproquement, si $c \models'_M \alpha$, alors $s_p(\downarrow_p c) \models'_M \alpha$. Il suffit alors de prendre $c' = s_p(\downarrow_p c)$ d'appliquer l'hypothèse d'induction pour conclure.
- Si $\phi = \alpha\ \mathcal{U}_p\ \beta$, l'équivalence de $c \models_M \phi$ et $c \models'_M \phi$ n'est plus qu'une question de notations.

□

Désormais, on notera donc *indifféremment* $\models_M$ quelque soit l'interprétation utilisée.

4.2.4 Localisation des formules

Comme on peut le constater sur la définition de l'interprétation $\models'_M$, la satisfaction d'une formule en une configuration c dépend principalement des i-vues de c. Afin de préciser cette constatation, nous allons définir la notion de *localisation* d'une formule de TrPTL.

Définition 14 (Localisation) *On appelle* localisation *l'application* loc *de l'ensemble des formules Φ de TrPTL dans l'ensemble des processus $\mathcal{P}$ définie inductivement par :*

- $\mathrm{loc}(q@p) = \mathrm{loc}(@_p\alpha) = \mathrm{loc}(\alpha\, \mathcal{U}_p\, \beta) = \{p\}$.
- $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$.
- $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.

Cette définition mène à la proposition suivante, dont la démonstration est immédiate par induction dans l'interprétation $\models'_M$:

Proposition 4 *Soit ϕ une formule de TrPTL, $M = (F, V)$ un modèle, et $c \in C_F$ une configuration. Alors les propriétés suivantes sont satisfaites :*

- *Si* $\mathrm{loc}(\phi) = \{p\}$ *alors* $c \models_M \phi$ *ssi* $\downarrow_p c \models_M \phi$.
- *Plus généralement, si* $\mathrm{loc}(\phi) = P \subseteq \mathcal{P}$, *alors* $c \models_M \phi$ *ssi* $\downarrow_P c \models_M \phi$, *où* $\downarrow_P c = \bigcup_{p \in P} \downarrow_p c$.

La satisfaction de ϕ en c ne dépend ainsi que des p-vues de c pour lesquelles $p \in \mathrm{loc}(\phi)$.

4.3 Expressivité et complexité

Après avoir présenté la logique TrPTL, nous nous intéressons maintenant aux aspects complexité et expressivité de cette logique, afin de pouvoir évaluer son intérêt pratique et théorique.

4.3.1 Complexité

Un des attraits principaux de la logique TrPTL est la possibilité d'en transformer toute formule en un automate asynchrone de Büchi définissant le même langage (cf. Chapitre 3). Plus précisemment, Mukund et Thiagarajan ont montré le résultat suivant, qui améliore un résultat précédent de Thiagarajan :

Théorème 12 ([44, 33]) *Pour toute formule α de taille n de TrPTL on peut construire un automate asynchrone de Büchi $\mathcal{A}_\alpha$ pour l'alphabet (Σ, D) vérifiant :*

(i) α est initialement satisfaisable ssi $L(\mathcal{A}_\alpha) \neq \emptyset$;

(ii) Le nombre d'états locaux de $\mathcal{A}_\alpha$ est borné par $2^{O(n^2 log(n))}$.

Il s'ensuit que le problème de la satisfaisabilité d'une formule α de taille n peut être résolu en temps $2^{O(kn^2 log(n))}$ dans TrPTL, où k est le cardinal de $\mathcal{P}$. Par ailleurs, le problème de la satisfaisabilité dans la logique LTL (qui est PSPACE-complet [43]) peut être décrit par un problème de satisfaisabilité dans TrPTL(avec un unique processus), donc ce dernier problème est PSPACE-dur [44].

4.3.2 Expressivité

Nous nous penchons à présent sur le problème de l'expressivité de la logique TrPTL. On rappelle que FO($\leq$) désigne la logique du premier ordre sur les traces (cf. Chapitre 3). Mukund et Thiagarajan ont montré le résultat suivant :

Théorème 13 ([33]) *Pour toute formule α de TrPTL, il existe une formule $f(\alpha)$ de $FO(\leq)$ telle que, pour tout modèle M,*

$$\emptyset \models_M \alpha \text{ ssi } M \models^{FO} f(\alpha)$$

Ainsi, tout langage définissable dans TrPTL l'est également dans $FO(\leq)$. La logique du premier ordre est donc au moins aussi expressive que la logique temporelle TrPTL. L'assertion réciproque reste à présent un problème ouvert. Autrement dit, on ne sait pas si TrPTL est *expressivement complète*, i.e. a le même pouvoir d'expression que FO($\leq$).

Expression d'un opérateur global dans TrPTL Nous allons maintenant montrer qu'il est difficile d'exprimer des propriétés globales dans TrPTL. La suite de cette Section comprend un résultat énoncé dans [46] (la Proposition 5) dont nous détaillons ici la preuve et un résultat “corollaire” original (la Proposition 6) dont nous donnons également une preuve complète. Il s'agit de résultats qui ne présentent pas de grandes difficultés mais qui permettent d'éclaircir considérablement les utilisations potentielles de TrPTL en tant qu'outil de spécification.

Supposons qu'on dispose d'une famille de formules $(\alpha_p)_{p\in\mathcal{P}}$ telle que pour tout processus $p \in \mathcal{P}$, la localisation de α_p est $\{p\}$. On dit qu'une configuration c satisfait $(\alpha_p)_{p\in\mathcal{P}}$ si pour tout $p \in \mathcal{P}$, $\downarrow_p c$ satisfait α_p, c'est-à-dire (grâce à la Proposition 4) si c satisfait la conjonction des α_p. On ne dispose pas a priori dans TrPTL d'opérateurs exprimant qu'on peut toujours atteindre (à partir d'une configuration présente c), une configuration c' plus grande que c et telle que c' satisfasse $(\alpha_p)_{p\in\mathcal{P}}$. Nous allons néanmoins montrer qu'il est possible de dériver un tel opérateur à partir des opérateurs de TrPTL.

Pour cela, on définit les abréviations suivantes [33]:

$$\forall p \in \mathcal{P}, \forall \alpha, \diamond_p \alpha \stackrel{\text{def}}{=} \underline{tt}\, \mathcal{U}_p\, \alpha.$$

Ainsi, pour tout modèle $M = (F, V)$, et toute configuration $c \in C_F$, on a :

$$c \models_M \diamond_p \alpha \text{ ssi } \exists c' \in C_F^p, \downarrow_p c \subseteq c' \text{ et } c' \models_M \alpha.$$

La formule $\diamond_p \alpha$ permet donc d'exprimer qu'il existe une p-vue plus grande que (ou égale à) la p-vue courante qui satisfasse α.

Par ailleurs, pour tout sous-alphabet X_p de Σ_p, on notera

$$\alpha\, \mathcal{U}_i^{X_p}\, \beta \stackrel{\text{def}}{=} (\alpha \wedge \bigwedge_{a\in\Sigma_p\setminus X_p} \neg \,@\,_p\underline{tt})\, \mathcal{U}_p\, \beta$$

La formule $\alpha\, \mathcal{U}_p^X\, \beta$ signifie donc que l'on doit atteindre une p-vue vérifiant β partir de la p-vue courante, et que toutes les p-vues intermédiaires doivent non seulement vérifier α, mais aussi interdire toute action de Σ_p autre que celle de l'ensemble X_p. Comme toutes les actions de Σ_p sont dépendantes, cela signifie que la succession des p-vues se fait uniquement par des actions de X_p.

En combinant les abréviations précédentes, on notera

$$\diamond_p^{X_p} \alpha \stackrel{\text{def}}{=} \underline{tt}\, \mathcal{U}_p^{X_p}\, \alpha$$

Ainsi, $c \models_M \diamond_i^{X_p} \alpha$ ssi on peut atteindre une p-vue c' plus grande que $\downarrow_p c$ qui satisfasse α de sorte que toutes les actions du processus p entre $\downarrow_p c$ et c' soient dans X_p, c'est-à-dire

$$(c' \setminus \downarrow c) \cap E_p \subseteq X_p.$$

On cherche maintenant à exprimer l'atteignabilité d'une configuration c satisfaisant $(\alpha_p)_{p \in \mathcal{P}}$ à partir de la configuration initiale $\emptyset$. On identifie à partir d'ici l'ensemble $\mathcal{P}$ des processus avec un sous-ensemble $\{1, 2, \ldots, k\}$ d'entiers. On utilise l'intuition suivante : pour atteindre la configuration c dont chaque p-vue satisfait la propriété α_p, on peut effectuer en premier lieu toutes les actions de $\downarrow_1 c$. On vérifie alors que α_1 est satisfaite. Ensuite, il faut effectuer d'autres actions afin de pouvoir satisfaire α_2, mais sans utiliser d'actions de Σ_1, afin de ne pas modifier $\downarrow_1 c$, et ainsi de suite. On est donc amené à définir une partition de Σ de la façon suivante : pour tout processus $p \in \mathcal{P}$ on définit un alphabet dérivé $\Sigma'_p = \Sigma_p \setminus (\cup_{p' < p} \Sigma_{p'})$. Les Σ'_p forment alors une partition de Σ. On peut maintenant définir l'abréviation suivante, pour la famille de formules $(\alpha_p)_{p \in \mathcal{P}}$:

$$\diamond^1((\alpha_p)_{p \in \mathcal{P}}) \stackrel{\text{def}}{=} \diamond_1^{\Sigma'_1}(\alpha_1 \wedge \diamond_2^{\Sigma'_2}(\alpha_2 \wedge \diamond_3^{\Sigma'_3}(\alpha_3) \wedge \cdots \diamond_k^{\Sigma'_k}(\alpha_k)) \cdots)$$

Proposition 5 *Pour tout modèle $M = (F, V)$, et pour toute famille $(\alpha_p)_{p \in \mathcal{P}}$ telle que $\mathrm{loc}(\alpha_p) \subseteq \{p\}$, l'opérateur $\diamond^1$ vérifie la propriété suivante :*

$$\emptyset \models_M \diamond^1((\alpha_p)_{p \in \mathcal{P}}) \textit{ ssi } \exists c \in C_F, c \models_M (\alpha_p)_{p \in \mathcal{P}}.$$

Preuve. Si $k = 1$, toutes les actions de $\Sigma = \Sigma_1 = \Sigma'_1$ sont deux à deux dépendantes, donc la logique TrPTL se réduit à la logique LTL. La propriété est alors trivialement vérifiée.

Afin de simplifier les notations, on considère d'abord le cas où $k = 2$. Dans ce cas, toute configuration $c \in C_F$ se décompose en $c = \downarrow_1 c \cup c'_1$, où $c'_1 = c \setminus \downarrow_1 c$. Posons $c_1 = \downarrow_1 c$. Puisque tous les événements de $c \cap E_1$ sont dans c_1, on a $c'_1 \cap E_1 = \emptyset$, ce qui s'écrit encore $c'_1 \subseteq (E_2 \setminus E_1)$. Toutes les actions correspondant aux événements de c'_1 sont donc dans le sous-alphabet Σ'_2. Ainsi, toute configuration c est atteignable à partir de $\downarrow_1 c$ par des actions de Σ'_2. La propriété en découle immédiatement.

Dans le cas général, on décompose simplement toute configuration c en $c = \downarrow_1 c \cup c'_1$, puis on décompose c'_1 en $c'_1 = (c'_1 \cap (E_2 \setminus E_1)) \cup c'_2$, et on note

$c_2 = c'_1 \cap (E_2 \setminus E_1)$, et ainsi de suite... On arrive finalement à décomposer c en

$$c = c_1 \cup c_2 \cup \ldots \cup c_k$$

avec $c_1 = \downarrow_1 c$, et pour tout $i > 1$,

$$c_i \subseteq E_i \setminus (\cup_{j<i} E_j)$$

ce qui mène à la propriété désirée. □

Remarquons que l'opérateur $\diamond^1(\ldots)$ permet d'exprimer l'atteignabilité à partir de la configuration vide, mais pas à partir de n'importe quelle configuration. En fait, en raison de l'ordre choisi sur les processus, on a $\mathrm{loc}(\diamond^1(\ldots)) = \{1\}$, et par conséquent cet opérateur ne peut exprimer l'atteignabilité qu'à partir des configurations qui sont des 1-vues. Plus précisément, on peut montrer la propriété suivante :

$$c \models_M \diamond^1((\alpha_p)_{p\in\mathcal{P}}) \text{ ssi } \exists c' \mid \downarrow_1 c \subseteq c' \text{ et } c' \models_M (\alpha_p)_{p\in\mathcal{P}}.$$

Sur le modèle de $\diamond^1$, on peut facilement construire des opérateurs $\diamond^p$ pour tout $p \in \mathcal{P}$ qui satisfont

$$c \models_M \diamond^p((\alpha'_p)_{p'\in\mathcal{P}}) \text{ ssi } \exists c' \mid \downarrow_p c \subseteq c' \text{ et } c' \models_M (\alpha_{p'})_{p'\in\mathcal{P}}.$$

Pour exprimer l'atteignabilité à partir d'une configuration quelconque, on est amené à utiliser la conjonction des opérateurs $\diamond^p$. Posons à cet effet

$$\diamond((\alpha_{p'})_{p'\in\mathcal{P}}) \stackrel{\text{def}}{=} \bigwedge_{p\in\mathcal{P}} \diamond^p((\alpha_{p'})_{p'\in\mathcal{P}})$$

Cet opérateur vérifie la proposition suivante :

Proposition 6 *Soit une famille $(\alpha_p)_{p\in\mathcal{P}}$ de formules telles que pour tout $p \in \mathcal{P}$, $\mathrm{loc}(\alpha_p) \subseteq \{p\}$. Alors, pour tout modèle $M = (F, V)$ et toute configuration $c \in C_F$, la propriété suivante est satisfaite :*

$$c \models_M \diamond(\{\alpha_p\}_{p\in\mathcal{P}}) \textit{ ssi } \exists c', c \subseteq c' \textit{ et } c' \models_M (\alpha_p)_{p\in\mathcal{P}}.$$

Preuve. S'il existe c' telle que $c \subseteq c'$ et $c' \models_M \alpha_1 \ldots \alpha_k$, alors on a aussi $\downarrow_p c \subseteq c'$ pour tout p, donc c' est atteignable à partir de toutes les p-vues de c. La conjonction des opérateurs $\diamond^p$ est alors satisfaite par la configuration c.

Réciproquement, on suppose que $c \models_M \diamond(\ldots)$. Pour tout $p \in \mathcal{P}$ il existe donc une configuration c_p qui vérifie $\downarrow_p c \subseteq c_p$ et qui satisfait $(\alpha_{p'})_{p' \in \mathcal{P}}$. Posons $c' = \cup_{p \in \mathcal{P}} c_p$. Alors pour tout i on a $\downarrow_p c \subseteq \downarrow_p c_p \subseteq \downarrow_p d$ car $\downarrow_p$ est à la fois monotone et involutif. Ceci prouve que $c \subseteq c'$. Enfin, pour tout $p \in \mathcal{P}$ il existe un $p' \in \mathcal{P}$ tel que $\downarrow_p c' = \downarrow_p c_{p'}$. Comme $\mathrm{loc}(\alpha_{p'}) \subseteq \{p'\}$, on obtient $\downarrow_p c' \models_M \alpha_{p'}$. On en conclut que c' satisfait la formule $(\alpha_{p'})_{p' \in \mathcal{P}}$. $\square$

On voit que l'on peut ainsi exprimer l'atteignabilité “globale” à partir d'une configuration quelconque. Plus précisemment, il est trivial que toute formule α de TrPTL s'écrit comme une combinaison booléenne de formules dont la localisation ne contient qu'un unique processus. En particulier, on peut toujours mettre une formule α sous la forme

$$\alpha = \bigvee_{i=1}^{n} \bigwedge_{p \in \mathcal{P}} \alpha_{i,p}$$

avec $\mathrm{loc}(\alpha_{i,p}) = \{p\}$. On peut alors définir un opérateur $\Diamond$ comme suit :

$$\Diamond\alpha = \bigvee_{i=1}^{n} \diamond(\{\alpha_{i,p}\}_{p \in P}).$$

D'après la Proposition 6, cette opérateur vérifie alors la propriété suivante :

$$c \models_M \Diamond\alpha \text{ ssi } \exists c' \supseteq c, c' \models_M \alpha.$$

La taille de la formule $\Diamond\alpha$ est linéaire en fonction de celle de la forme normale disjonctive de α. Il se peut donc qu'elle soit exponentielle en fonction de celle de α.

L'intérêt de cette construction est de montrer que, malgré le caractère local de TrPTL, il est possible (bien que fastidieux) d'écrire un opérateur global du type *eventually* dans cette logique. On ne sait pas (mais on conjecture que ce n'est pas le cas) s'il est possible de faire des constructions similaires pour des opérateurs globaux *binaires* du type *until*.

4.4 Propriétés élémentaires des opérateurs

On s'intéresse maintenant à des propriétés plus élémentaires et plus générales des opérateurs temporels de TrPTL.

Ces opérateurs temporels présentent des propriétés de distributivité sur les connecteurs booléens $\vee$ et $\wedge$. Pour les exprimer, on recourt à la notion d'équivalence de formules définie comme suit :

Définition 15 (Équivalence dans TrPTL) *Soient α et β deux formules de TrPTL. On dit que α et β sont équivalentes (resp. initialement équivalentes), et on note $\alpha \equiv \beta$ (resp. $\alpha \equiv_{in.} \beta$) ssi pour tout modèle $M = (F, V)$, et pour toute configuration $c \in C_F$, on a $c \models_M \alpha$ ssi $c \models_M \beta$ (resp. $\emptyset \models_M \alpha$ ssi $\emptyset \models_M \beta$).*

Il est immédiat que deux formules équivalentes sont aussi initialement équivalentes.

Proposition 7 *Soient α, β, et γ des formules de TrPTL, soit $p \in \mathcal{P}$ et soit $a \in \Sigma_p$. Alors les équivalences suivantes sont vérifiées:*

(1) $@_p(\alpha \vee \beta) \equiv @_p\alpha \vee @_p\beta$

(2) $@_p(\alpha \wedge \beta) \equiv @_p\alpha \wedge @_p\beta$

(3) $(\alpha \wedge \beta)\, \mathcal{U}_p\, \gamma \equiv (\alpha\, \mathcal{U}_p\, \gamma) \wedge (\beta\, \mathcal{U}_p\, \gamma)$

(4) $\alpha\, \mathcal{U}_p\, (\beta \vee \gamma) \equiv (\alpha\, \mathcal{U}_p\, \beta) \vee (\alpha\, \mathcal{U}_p\, \gamma)$

Les opérateurs temporels locaux de TrPTL suivent donc les même règles de distributivité que les opérateurs correspondants dans la logique LTL, vis-à-vis des connecteurs booléens.

4.5 Alphabet non-connexe

On rappelle que le théorème de Mezei montre que tout langage reconnaissable d'un produit $M_1 \times M_2$ de deux monoïdes est une union finie de produits du type $L_1 \times L_2$ où pour $i = 1, 2$, L_i est un langage reconnaissable de M_i [2]. Ce théorème reste valable lorsque l'on considère des langages "sans-étoile" et non plus des langages "reconnaissables". Si TrPTL était une logique temporelle expressivement complète, alors on aurait une équivalence entre langages sans-étoile et langages définissables dans TrPTL. La version "sans-étoile" du théorème de Mezei impliquerait alors que tout langage défini par une formule de TrPTL sur un alphabet non-connexe est une union finie de langages définis par des formules de TrPTL telles que chaque formule n'implique qu'une seule des composantes connexes de l'alphabet. Même si la complétude de TrPTL (au sens de son expressivité) demeure un problème ouvert, nous montrons dans la suite de ce chapitre que cette logique temporelle permet bien une telle décomposition des formules. De plus, cette décomposition est *effective.*

Afin de simplifier l'exposé de ce résultat, nous nous concentrons dans un premier temps sur le cas d'un alphabet de dépendance ne possédant que deux composantes connexes. Dans ce cas, on peut partitionner Σ en deux sous alphabets Σ_A et Σ_B. Notons D_A et D_B les restrictions respectives de D à Σ_A et à Σ_B. En termes de graphes, (Σ, D) est alors l'union de (Σ_A, D_A) et $(\Sigma_B.D_B)$. On dira par la suite que Σ_A et Σ_B sont des *composantes* de Σ.

Remarquons que pour processus $p \in \mathcal{P}$, toutes les actions de Σ_p sont dans une même composante, puisqu'elles sont deux-à-deux dépendantes. Ainsi, on peut associer aux composantes Σ_A et Σ_B de Σ des *composantes* de l'ensemble des processus $\mathcal{P}$:

$$\mathcal{P}_A = \{p \in \mathcal{P} \mid \Sigma_p \subseteq \Sigma_A\}$$

$$\mathcal{P}_B = \{p \in \mathcal{P} \mid \Sigma_p \subseteq \Sigma_B\}$$

A partir des composantes de l'alphabet distribué, on peut définir deux sous-logiques de TrPTL, à l'aide des restrictions syntaxiques suivantes :

Définition 16 *On appelle $TrPTL_A$ (respectivement $TrPTL_B$) l'ensemble des formules ϕ de TrPTL telles que tous les processus apparaissant dans ϕ soient dans $\mathcal{P}_A$ (resp. dans $\mathcal{P}_B$).*

Une formule est donc dans TrPTL_A si et seulement si on a $\mathrm{loc}(\psi) \subseteq \mathcal{P}_A$ pour toute sous-formule ψ de ϕ.

Le théorème suivant forme le résultat principal de ce chapitre. Il s'agit d'un théorème de décomposition effective des formules de la logique TrPTL, présenté dans [31].

Théorème 14 *Si (Σ, D) présente deux composantes connexes (Σ_A, D_A) et (Σ_B, D_B), alors toute formule ϕ de TrPTL est* initialement *équivalente à une disjonction finie de formules du type $\alpha_A \wedge \alpha_B$, où $\alpha_A \in TrPTL_A$ et $\alpha_B \in TrPTL_B$. De plus, ces formules peuvent effectivement être construites à partir de ϕ.*

Lorsque l'alphabet de dépendance possède un nombre quelconque de composantes connexes, le théorème précédent se généralise comme suit :

Théorème 15 *Si* (Σ, D) *est un alphabet à* m *composantes connexes notées* (Σ_1, D_1),...,(Σ_m, D_m)*, alors toute formule* ϕ *de TrPTL est initialement équivalente à une disjonction finie de formules du type* $\alpha_1 \wedge \ldots \wedge \alpha_m$*, où pour tout* $j \leq m$*,* α_j *est une formule n'utilisant que des processus de* $\mathcal{P}_J$*. De plus, cette décomposition est effective.*

Dans la suite de cette section nous présentons la preuve complète du Théorème 14.

Afin de faciliter cette démonstration, on introduit un nouvel opérateur temporel, noté R pour "Reset", dans la logique TrPTL. La sémantique de cet opérateur est la suivante : pour toute formule α de TrPTL, pour tout modèle $M = (F, V)$ et toute configuration $c \in C_F$ de ce modèle,

$$c \models_M R\alpha \text{ ssi } \emptyset \models_M \alpha.$$

L'opérateur R permet donc de se débarrasser de la configuration présente et de poursuivre l'interprétation des formules en repartant de la configuration vide. Il s'agit donc d'un opérateur de "passé" d'un type tout-à-fait particulier. On conviendra de définir la localisation d'une formule de type $R\alpha$ comme suit :

$$\text{loc}(R\alpha) = \text{loc}(\alpha).$$

On notera TrPTL$^+$ la logique TrPTL enrichie de l'opérateur R. On ne se soucie pas pour l'instant d'un éventuel accroissement d'expressivité dû à l'opérateur R. De même, on notera TrPTL$_A^+$ et TrPTL$_B^+$ les deux sous-logiques enrichies de R.

On appelle *formule de décomposition* toute formule de TrPTL$^+$ de la forme

$$\alpha_A \wedge R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B$$

où α_A et α'_A sont des formules de TrPTL$_A$, et où α_B et α'_B sont des formules de TrPTL$_B$.

Remarquons que le Théorème 14 considère des formules "initialement équivalentes". Comme nous allons procéder par induction sur la nature des formules à décomposer, il nous faut exhiber une décomposition plus forte, dans laquelle les formules sont "équivalentes". Nous allons donc prouver le théorème suivante, qui implique immédiatement le Théorème 14.

Théorème 16 *Si (Σ, D) présente deux composantes connexes (Σ_A, D_A) et (Σ_B, D_B), alors toute formule ϕ de TrPTL est équivalente à une disjonction finie de formules de décomposition. De plus, ces formules peuvent effectivement être construites à partir de ϕ.*

L'alphabet (Σ, D) ayant deux composantes connexes, il en est de même pour les traces sur (Σ, D) en général. En effet, pour tout modèle $M = (F, V)$, l'ensemble des événements E peut être décomposé en $E = E_A \cup E_B$, où $E_A = \{e \in E \mid \lambda(e) \in \Sigma_A\}$ et $E_B = \{e \in E \mid \lambda(e) \in \Sigma_B\}$. Ces deux ensembles sont évidemment disjoints, mais ils ne sont pas nécessairement non-vides. Toute configuration $c \in C_F$ se décomposera donc en deux composantes disjointes c_A et c_B, chaque composante étant elle-même une configuration. La proposition suivante est un cas particulier de la Proposition 4.

Proposition 8 *Soit $i \in \mathcal{P}_A$ et soit $c \in C_F$ une configuration d'un modèle $M = (F, V)$. Soit une formule α de TrPTL telle que $\mathrm{loc}(\alpha) \subseteq \mathcal{P}_A$. Alors*

$$c \models_M \alpha \textit{ ssi } c_A \models_M \alpha.$$

Notons qu'on ne demande pas que α soit dans TrPTL_A, mais seulement que sa localisation soit un sous-ensemble de $\mathcal{P}_A$.

On s'intéresse maintenant à l'action des opérateurs temporels de TrPTL sur les formules de décomposition.

Proposition 9 *Soit $p \in \mathcal{P}_A$, soit $a \in \Sigma_p$, soient α_A et α'_A des formules de $TrPTL_A$, et soient α_B et α'_B des formules de $TrPTL_B$. Soit de plus β une formule quelconque de $TrPTL^+$. Alors on a les équivalences suivantes :*

$$@_p(\alpha_A \wedge R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B) \equiv @_p\alpha_A \wedge R\alpha'_A \wedge R(\alpha_B \wedge \alpha'_B) \quad (4.1)$$
$$\beta\, \mathcal{U}_p\, (\alpha_A \wedge R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B) \equiv \beta\, \mathcal{U}_p\, \alpha_A \wedge R\alpha'_A \wedge R(\alpha_B \wedge \alpha'_B) \quad (4.2)$$
$$(\alpha_A \wedge R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B)\, \mathcal{U}_p\, \beta \equiv \alpha_A \mathcal{U}_p \beta \wedge (\beta \vee R(\alpha'_A \wedge \alpha_B \wedge \alpha'_B)) \quad (4.3)$$

Afin de faciliter la preuve de la Proposition 9, nous introduisons le lemme suivant :

Lemme 1 *Soit $p \in \mathcal{P}_A$, soit α_B une formule de $TrPTL_B$, et soit c une configuration quelconque d'une trace F. Alors*

$$\downarrow_p c \models_M \alpha_B \textit{ ssi } \emptyset \models_M \alpha_B.$$

Preuve. D'après la Proposition 4, $\downarrow_p c \models_M \alpha_B$ ssi $\downarrow_{\mathcal{P}_B} (\downarrow_p c) \models_M \alpha_B$. Or $A \times B \subseteq I$, donc $\downarrow_{\mathcal{P}_B}(\downarrow_p c) = \emptyset$. □

On peut maintenant entamer la preuve de la Proposition 9.

Preuve. Posons $\alpha = \alpha_A \wedge R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B$.

Preuve de 4.1 : En utilisant la distributivité de $@_p$ sur la conjonction, on obtient que $@_p\alpha$ équivaut à $@_p\alpha_A \wedge @_pR\alpha'_A \wedge @_p\alpha_B \wedge @_pR\alpha'_B$. Or $@_p\alpha_B$ équivaut à $@_p\underline{tt} \wedge R\alpha_B$ puisque $\alpha_B \in \mathrm{TrPTL}_B$. De même, $@_pR\alpha'_B$ équivaut à $@_p\underline{tt} \wedge R\alpha'_B$. Enfin, $@_pR\alpha'_A$ équivaut à $@_p\underline{tt} \wedge R\alpha'_A$. Comme de plus $@_p\alpha_A$ implique $@_p\underline{tt}$, on obtient l'équivalence voulue, l'opérateur R étant distributif sur la conjonction.

Preuve de 4.2 : Soit c une configuration d'un modèle $M = (F, V)$. Puisque $p \in \mathcal{P}_A$, on a $c \models_M \beta\, \mathcal{U}_p\, \alpha$ ssi $c_A \models_M \beta\, \mathcal{U}_p\, \alpha$, i.e. ssi il existe une p-vue c'_p telle que $\downarrow_p(c_A) \subseteq c'_p$ et telle que $c'_p \models_M \alpha$, et si pour toute p-vue c''_p telle que $\downarrow_p(c_a) \subseteq c''_p \subset c'_p$ on ait $c''_p \models_M \beta$. Or $c'_p \models_M \alpha$ équivaut à $c'_p \models_M \alpha_A$ et $\emptyset \models_M \alpha'_A \wedge \alpha_B \wedge \alpha'_B$ en utilisant le Lemme 1. On a donc $c \models_M \beta\, \mathcal{U}_p\, \alpha$ ssi $c \models_M \beta\, \mathcal{U}_p\, \alpha_A$ et $\emptyset \models_M \alpha'_A \wedge \alpha_B \wedge \alpha'_B$, ce qui mène directement à l'équivalence recherchée.

Preuve de 4.3 : La distributivité à gauche de $\mathcal{U}_p$ sur $\wedge$ montre que la formule $\alpha\, \mathcal{U}_p\, \beta$ est équivalente à $\alpha_A\, \mathcal{U}_p\, \beta \wedge (R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B)\, \mathcal{U}_p\, \beta$. Or, lorsqu'on exprime la formule $(R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B)\, \mathcal{U}_p\, \beta$ selon la définition du $\mathcal{U}_p$, la condition sur les p-vues c''_p, $c''_p \models_M (R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B)$ devient $\emptyset \models_M (\alpha'_A \wedge \alpha_B \wedge \alpha'_B)$ en utilisant le Lemme 1. Cette propriété ne dépend donc pas de c''_p. Elle peut donc s'écrire $c \models_M R(R\alpha'_A \wedge \alpha_B \wedge R\alpha'_B)$. Néanmoins, si β est immédiatement vérifiée en c, on n'a plus besoin de satisfaire $R(\alpha'_A \wedge \alpha_B \wedge \alpha'_B)$. Ceci conduit à l'équivalence entre $\alpha\, \mathcal{U}_p\, \beta$ et $\alpha_A\, \mathcal{U}_p\, \beta \wedge (\beta \vee R(\alpha'_A \wedge \alpha_B \wedge \alpha'_B))$. □

Les deux lemmes techniques suivants nous seront nécessaire afin de démontrer le Théorème 16.

Lemme 2 *Soient α_A une formule de $TrPTL^+_A$, α_B une formule de $TrPTL^+_B$, et β une formule quelconque de $TrPTL^+$. Soit $p \in \mathcal{P}_A$. Alors*

$$(\alpha_A \vee \alpha_B)\, \mathcal{U}_p\, \beta \equiv (\alpha_A\, \mathcal{U}_p\, \beta) \vee (R\alpha_B \wedge \underline{tt}\, \mathcal{U}_p\, \beta).$$

Preuve. Soit $M = (F, V)$ un modèle, soit $c \in C_F$ une configuration. Alors $c \models_M (\alpha_A \vee \alpha_B)\, \mathcal{U}_p\, \beta$ ssi il existe une p-vue c' plus grande que $\downarrow_p c$ telle que $c' \models_M \beta$, et si pour toute p-vue c'', si $\downarrow_p c \subseteq c'' \subset c'$ alors $c'' \models_M \alpha_A$ ou $c'' \models_M \alpha_B$. Comme α_B est dans TrPTL^+_B, cette dernière condition s'écrit

$\emptyset \models_M \alpha_B$ en utilisant le Lemme 1, et ne dépend donc pas de c''. Ainsi, si $\emptyset \models_M \alpha_B$, on n'a jamais besoin de vérifier $c'' \models_M \alpha_A$. Si au contraire $\emptyset \not\models_M \alpha_B$, alors toutes les p-vues intermédiaires c'' doivent satisfaire α_A. On obtient ainsi l'équivalence souhaitée. $\square$

Lemme 3 *Soient γ_A, γ'_A et β_A des formules de TrPTL$_A$, et soit $p \in \mathcal{P}_A$. Alors*

$$(\gamma_A \vee R\gamma'_A)\, \mathcal{U}_p\, \beta_A \equiv \gamma_A\, \mathcal{U}_p\, \beta_A \vee (R\gamma'_A \wedge \underline{tt}\, \mathcal{U}_p\, \beta_A).$$

Preuve. Soit $M = (F, V)$ un modèle, soit $c \in C_F$ une configuration. Alors $c \models_M (\gamma_A \vee R\gamma'_A)\, \mathcal{U}_p\, \beta_A$ ssi il existe une p-vue c' plus grande que $\downarrow_p c$ telle que $c' \models_M \beta_A$ et si pour toute p-vue c'', si $\downarrow_p c \subseteq c'' \subset c'$ alors $c'' \models_M \gamma_A \vee R\gamma'_A$. Or $c'' \models_M \gamma_A \vee R\gamma'_A$ ssi $c'' \models_M \gamma_A$ ou $\emptyset \models_M \gamma'_A$ d'après la définition de l'opérateur R. Comme la condition $\emptyset \models_M \gamma'_A$ ne dépend pas de c'', elle est vraie quelle que soit c'' ou fausse quelle que soit c''. Si elle est vrai, i.e si $c \models_M R\gamma'_A$, alors il suffit de vérifier $c \models_M \underline{tt}\, \mathcal{U}_p\, \beta_A$. En revanche, si $c \not\models_M R\gamma'_A$, on doit vérifier $c \models_M \gamma_A\, \mathcal{U}_p\, \beta_A$. D'où l'équivalence recherchée. $\square$

Nous pouvons maintenant aborder la preuve du Théorème 16. On rappelle que l'on cherche à décomposer toute formule ϕ de TrPTL en une disjonction finie équivalente de formules de décomposition.

Preuve. On raisonne par récurrence sur la structure de la formule ϕ.

- Si $\phi = q@p$, alors ϕ est déjà une formule de décomposition (avec des termes triviaux).

- Si $\phi = \alpha \vee \beta$, l'hypothèse de récurrence appliquée à α et à β permet de conclure.

- Si $\phi = \neg\alpha$, alors α est par hypothèse équivalente à une disjonction finie de formules de décomposition. En distribuant la négation, on peut réarranger les termes de ϕ en une disjonction de nouvelles formules de décomposition.

- Si $\phi = @_p\alpha$, on utilise la distributivité de $@_p$ sur la disjonction, puis on applique l'équivalence 4.1 de la Proposition 9.

- Si $\phi = \alpha\ \mathcal{U}_p\ \beta$, on applique d'abord l'hypothèse de récurrence à la formule β. L'opérateur $\mathcal{U}_p$ étant distributif à droite sur la disjonction, on est ramené à décomposer des formules du type $\alpha\ \mathcal{U}_p\ \beta'$, où β' est une formule de décomposition. On applique alors l'équivalence 4.2 de la Proposition 9, et on est ramené au cas des formules du type $\alpha\,\mathcal{U}_p\ \beta_A$, où β_A est une formule de TrPTL_A. L'hypothèse de récurrence montre alors que α est équivalente à une disjonction de formules de décomposition. On réarrange alors les termes de cette disjonction pour obtenir une conjonction de formules du type $\alpha_A \vee \alpha_B$, où α_A est de la forme $\cup_j \alpha_{A,j} \wedge R\alpha'_{A,j}$, avec $\alpha_{A,j}, \alpha'_{A,j}$ des formules de TrPTL_A, et où α_B est de la forme $\cup_j \alpha_{B,j} \wedge R\alpha'_{B,j}$, avec $\alpha_{B,j}, \alpha'_{B,j}$ des formules de TrPTL_B. En utilisant la distributivité à gauche de $\mathcal{U}_p$ sur la conjonction, on obtient une conjonction de formules du type $(\alpha_A \vee \alpha_B)\,\mathcal{U}_p\ \beta_A$. On utilise alors le Lemme 2 pour décomposer de telles formules en $\alpha_A\,\mathcal{U}_p\ \beta_A \vee (R\alpha_B \wedge \underline{tt}\,\mathcal{U}_p\ \beta_A)$. Mais $R\alpha_B$ s'écrit encore $\cup_j R(\alpha_{B,j} \wedge \alpha'_{B,j})$. Pour traiter la formule $\alpha_A\,\mathcal{U}_p\ \beta_A$, on écrit α_A comme une conjonction de formules du type $\gamma_A \vee R\gamma'_A$, et on applique à nouveau la distributivité à droite de $\mathcal{U}_p$ sur $\wedge$, pour obtenir des formules du type $(\gamma_A \vee R\gamma'_A)\,\mathcal{U}_p\ \beta_A$. Le Lemme 3 montre alors qu'une formule de ce type est équivalente à la formule suivante : $\gamma_A\,\mathcal{U}_p\ \beta_A \vee (R\gamma'_A \wedge \underline{tt}\,\mathcal{U}_p\ \beta_A)$. En regroupant tous les termes de cette décomposition, on obtient une combinaison booléenne positive (i.e. sans négation) de formules de décomposition, que l'on peut réarranger en une grande disjonction d'autres formules de décomposition.

□

Exemple 9 *Considérons l'alphabet distribué non-connexe $\{\Sigma_1, \Sigma_2\}$, avec $\Sigma_1 = \{a, b\}$ et $\Sigma_2 = \{d, e\}$, et considérons la formule*

$$\phi = (p@2)\,\mathcal{U}_1\ (q@2).$$

Alors, en appliquant la méthode énoncée dans la preuve du Théorème 16, on écrit d'abord ϕ sous la forme

$$\phi \equiv (p@2)\,\mathcal{U}_1\ \underline{tt} \wedge R(q@2),$$

en utilisant l'équivalence 4.2 de la Proposition 9. Puis on utilise le Lemme 2 pour écrire

$$\phi \equiv (\underline{tt}\,\mathcal{U}_1\ \underline{tt} \vee (R(p@2) \wedge \underline{tt}\,\mathcal{U}_1\ \underline{tt})) \wedge R(q@2).$$

A ce stade la formule est déjà décomposée, mais en remplaçant les $\underline{tt}\ \mathcal{U}_1\ \underline{tt}$ par $\underline{tt}$ on obtient :

$$\phi \equiv R(q@2).$$

Autrement dit, ϕ est initialement équivalente à $q@2$. En effet, on a vu sur un exemple précédent que ϕ spécifiait un until *sur les configurations de synchronisation. Comme ici les processus* 1 *et* 2 *ne communiquent pas, la seule configuration de synchronisation est la configuration vide, et donc le "p* until *q" se résume à "q".*

4.6 Applications

La preuve du Théorème 16 montre que la décomposition d'une formule de TrPTL en formules de décomposition peut engendrer une explosion combinatoire (dans la disjonction équivalente). Afin d'éviter cette explosion, on peut définir une classe de formules de TrPTL dans laquelle toute formule se décompose avec une complexité faible. On définit cette classe, notée Φ_s de la façon suivante :

Définition 17 *Soit Φ_s le plus petit ensemble de formules de TrPTL satisfaisant les conditions suivantes :*

- *Pour tout $q \in AP$ et pour tout $p \in \mathcal{P}$, $q@p \in \Phi_s$.*
- *Si $\alpha, \beta \in \Phi_s$, alors $\alpha \vee \beta \in \Phi_s$.*
- *Si $\alpha \in \Phi_s$, alors pour tout $p \in \mathcal{P}$ et pour tout $a \in \Sigma_p$, $@_p\alpha \in \Phi_s$.*
- *Si $\alpha \in TrPTL_A$ et $\beta \in \Phi_s$ alors pour tout $p \in \mathcal{P}_A$, $\alpha\, \mathcal{U}_p\, \beta \in \Phi_s$.*
- *Si $\alpha \in TrPTL_B$ et $\beta \in \Phi_s$ alors pour tout $p \in \mathcal{P}_B$, $\alpha\, \mathcal{U}_p\, \beta \in \Phi_s$.*

Les formules de Φ_s sont dites décomposables.

Notons que l'on interdit dans Φ_s l'emploi de la négation, et que l'on impose certaines restrictions syntaxiques sur l'emploi des opérateurs $\mathcal{U}_p$. L'intérêt des formules *décomposables* réside dans la proposition suivante :

Proposition 10 *Soit $\gamma \in \Phi_s$ une formule décomposable de taille n comportant d disjonctions. Alors γ peut être décomposée effectivement en une disjonction de d formules de décomposition de taille au plus $(n - d)$ et ne contenant aucune disjonction.*

Preuve. Puisque $@_p(\alpha \vee \beta) = @_p\text{alph} \vee @_p\beta$ et $\alpha\, \mathcal{U}_p\, (\beta \vee \theta) = \alpha\, \mathcal{U}_p\, \beta \vee \alpha\, \mathcal{U}_p\, \theta$, on peut faire passer toutes les disjonctions de γ à l'extérieur des opérateurs temporels. La proposition en découle immédiatement. $\square$

Exemple 10 *Soit* $\gamma = q@1\, \mathcal{U}_1\, (@_1(q@2\, \mathcal{U}_2\, q'@1))$*, avec* $1 \in \mathcal{P}_A$ *et* $2 \in \mathcal{P}_B$*. Alors* $\gamma \in \Phi_s$ *et la décomposition de* γ *est la suivante :*

$$\gamma \equiv p@1\, \mathcal{U}_1\, (@_1\underline{tt}) \wedge Rp@2 \wedge Rp'@1.$$

On voit sur cet exemple que la décomposition permet de "casser" certaines imbrications d'opérateurs temporels.

Lorsqu'on s'intéresse à l'équivalence *initiale* des formules, on peut effacer tous les opérateurs R de la décomposition obtenue. Intuitivement, la décomposition d'une formule de Φ_s consiste à couper la formule à chaque fois que deux opérateurs temporels successivement imbriqués réfèrent à des composantes connexes distinctes.

4.7 Conclusion

Dans ce chapitre nous avons démontré un théorème de décomposition des formules de TrPTL dans le cas d'un alphabet de dépendance non-connexe. Il s'agit principalement d'un résultat théorique, d'une certaine façon analogue au théorème de Mezei, mais qui peut également permettre dans certains cas une accélération drastique de la vérification en découpant la formule initiale en une combinaison booléenne de formules plus petites. En outre, cette étude permet de mettre en évidence certaines faiblesses de la logique TrPTL; en particulier, on voit dans le cadre d'un alphabet non-connexe que l'alternance d'opérateurs temporels appartenant à l'une puis à l'autre des composantes connexes se traduit par un "retour à l'origine" (utilisation de l'opérateur R). Dans le cadre d'un alphabet quelconque, cette alternance peut se traduire par un "retour dans le passé" qui rend difficile toute interprétation intuitive des formules de TrPTL. Nous pouvons donc conclure ainsi sur les utilisations potentielles de TrPTL : cette logique permet d'écrire des spécifications utiles concernant un processus isolé au sein de l'ensemble des processus, car sa sémantique est alors identique à celle de LTL pour les états locaux de ce processus; en revanche, la spécification de comportements globaux d'un système distribué se fait avec difficulté dans TrPTL, car alors l'interaction des différents opérateurs temporels est peu intuitive.

Notre décomposition prouve justement que, dans le cadre d'un alphabet non-connexe, on peut se limiter à spécifier chaque composante connexe individuellement (ce qui est intuitivement évident puisque deux composantes distinctes ne communiquent pas), car toute spécification mêlant les différentes composantes se décompose *in fine* en une combinaison booléenne de spécifications individuelles.

Chapitre 5

Une procédure de décision modulaire

5.1 Introduction

Nous nous intéressons à présent aux problèmes liés à la vérification dans le cadre de la logique LTrL[46], et notamment à la satisfaisabilité d'une formule de cette logique. Rappelons qu'il s'agit d'une logique temporelle *globale* sur les traces, dont le pouvoir d'expression est celui de la théorie du premier ordre FO($\leq$).

En termes de complexité, Walukiewicz a montré que le problème de la satisfaisabilité d'une formule de LTrL est NON-ELEMENTAIRE [50]. Plus précisemment, soit *tour* la fonction "tour d'exponentielles" définie par $tour(0, m) = m$ et pour tout entier k, $tour(k+1, m) = 2^{tour(k,m)}$. Pour un entier k fixé, la fonction $m \mapsto tour(k, m)$ est appelée "tour d'exponentielles de hauteur m". La fonction *tour* a donc l'allure suivante :

$$tour(n, x) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^x}}}}}}, n \text{ étant la hauteur de l'empilement.}$$

Pour tout entier n, il existe une formule ψ_n de LTrL telle que décider de la satisfaisabilité de ψ_n ne peut pas se faire en temps $tour(n-1, x)$, où x est la taille de la formule ψ_n. Ceci montre que le problème de la satisfaisabilité dans LTrL ne peut se résoudre en temps borné par une fonction tour d'exponentielles de hauteur fixée. Une telle complexité est dite non-élémentaire.

Par ailleurs, ce problème peut être décidé en temps $tour(n, 2)$ pour toute formule de taille n (cette fois la tour d'exponentielles est de hauteur la taille de la formule). En effet, toute formule de LTrL de taille n peut être traduite en une formule de FO($\leq$) de taille $O(n)$, et on peut alors ramener le problème de la satisfaisabilité d'une formule de FO($\leq$) à celui d'une formule de la logique du premier ordre sur les mots, problème non-élémentaire qui admet un algorithme en temps $tour(n, 2)$ pour une formule de taille n [37].

Une telle complexité paraît a priori rebutante, mais ce n'est pas toujours le cas dans la pratique. En effet, il s'agit d'une complexité théorique "dans le pire cas", et on peut espérer que la complexité moyenne d'une formule de LTrL soit plus faible; plus précisemment, la complexité est non-élémentaire en fonction de deux facteurs : la profondeur d'imbrication "à gauche" des opérateurs *until* (c'est-à-dire des emboîtements du type $((\alpha\,\mathcal{U}\,\beta)\,\mathcal{U}\,\gamma)\,\mathcal{U}\,\delta)$ et les négations. Comme il est rare décrire des formules de spécification utiles dont la profondeur d'imbrication à gauche des *until* soit supérieure à 2, on peut espérer obtenir dans la plupart des cas une complexité relativement raisonnable. Par ailleurs, on sait que certains outils tels Mona [24] permettent d'obtenir des résultats sur des problèmes dont la complexité est non-élémentaire : Mona est un outil de spécification et de vérification en logique monadique du second ordre (sur les mots), qui permet notamment de spécifier des circuits *hardware* paramétrés (par exemple, un additioneur paramétré) grâce à son grand pouvoir d'expression. Malgré le caractère non-élémentaire de cette logique, l'outil Mona permet d'obtenir des résultats intéressants en pratique.

Dans ce chapitre nous proposons une procédure de décision *directe* pour la logique temporelle LTrL, c'est-à-dire une procédure qui permet de traduire toute formule de LTrL en un automate (de Büchi) qui reconnaît le même langage. Décider de la satisfaisabilité d'une formule revient alors à décider du vide pour un automate de Büchi, ce qui peut se faire en temps polynômial en fonction de la taille de l'automate (la taille étant mesurée par le nombre d'états et de transitions). La logique LTrL étant non-élémentaire, la taille de l'automate obtenu peut être non-élémentaire en fonction de la taille de la formule d'origine. Plus précisemment, on construira, pour une formule ψ donnée, un automate de Büchi sur les mots qui reconnaît exactement les linéarisations des modèles de ψ. De plus, notre traduction est *modulaire*, c'est-à-dire qu'elle se fait par induction sur la structure de la formule, ce qui permet de réutiliser des automates construit auparavant lorsque l'on cherche à traduire une formule dont une ou des sous-formules ont déjà été traduites. Le choix de cette approche modulaire va nous conduire à construire non pas un seul automate, mais une famille (finie) d'automates pour chaque formule. Comme nous le verrons par la suite, les deux cas qui provoquent structurellement une explosion combinatoire sont la négation et l'opérateur *until*.

Ces résultats ont fait l'objet d'une publication lors de la conférence MFCS à Brno, en août 1998 [12].

5.2 Syntaxe et sémantique de LTrL

La logique LTrL, proposée en 1997 par Thiagarajan et Walukiewicz [46], est une logique temporelle *globale* sur les traces : ses opérateurs sont fondées sur ceux de LTL [39], mais contrairement à ceux de TrPTL, leur portée s'étend aux configurations d'une trace réelle, et non aux vues partielles de ces configurations (cf Chapitre 3).

L'ensemble des formules de la logique temporelle LTrL est défini inductivement par :

$$\text{LTrL} ::= \underline{tt} \mid \neg\alpha \mid \alpha \vee \beta \mid \textcircled{a}\alpha \mid \alpha\,\mathcal{U}\,\beta \mid \underline{a}, a \in \Sigma$$

Un modèle pour LTrL est simplement une trace réelle $u = (E, \leq, \lambda)$. Les formules de LTrL sont interprétées pour une trace donnée u, et en une configuration donnée $c \in C_u$ de cette trace.

La sémantique d'une formule de LTrL est définie inductivement comme suit, pour toute trace u et toute configuration $c \in C_u$:

- $u, c \models \underline{tt}$.
- $u, c \models \underline{a}$ ssi il existe un événement maximal e dans c (pour l'ordre partiel $\leq$) tel que $\lambda(e) = a$.
- $u, c \models \neg\alpha$ ssi $u, c \not\models \alpha$.
- $u, c \models \alpha \vee \beta$ ssi $u, c \models \alpha$ ou $u, c \models \beta$.
- $u, c \models \textcircled{a}\alpha$ ssi il existe une configuration $c' \in C_u$ telle que $c \xrightarrow{a} c'$ et $u, c' \models \alpha$.
- $u, c \models \alpha\,\mathcal{U}\,\beta$ ssi il existe une configuration $c' \in C_u$ telle que $c \subseteq c'$ et $u, c' \models \beta$ et si de plus, pour toute configuration $c'' \in C_u$, si $c \subseteq c'' \subset c'$ alors $u, c'' \models \alpha$.

Les formules $\underline{a}$, pour $a \in \Sigma$, jouent le rôle de formules atomiques. Elles permettent de décrire la configuration présente c, à travers ses événements maximaux. On les appelle pour cette raison des formules de présent. Par opposition, les opérateurs ⓐ et $\mathcal{U}$ sont des opérateurs dits de pur futur, puisque leur interprétation ne dépend que des configurations postérieures à la configuration présente.

On remarquera que la définition des opérateurs temporels ⓐ et $\mathcal{U}$ est strictement similaire à celle des opérateurs correspondants dans LTL, en remplaçant la notion de préfixe par celle de configuration. Contrairement à ce qui se passe pour le logique TrPTL, on s'intéresse ici à toutes les configurations, et pas seulement aux configurations particulières que sont les vues partielles.

Exemple 11 *Soit $\phi = ⓐ_p\alpha$ une formule de TrPTL, où $a \in \Sigma_p$ et soit $\psi = ⓐ\beta$ une formule de LTrL. Soit c une configuration d'une trace $F = (E, \leq, \lambda)$. Dans TrPTL, l'assertion $c \models \phi$ signifie que la p-vue c'_p qui succède à la p-vue courante $c_p = \downarrow_p c$ doit vérifier $c'_p \models \alpha$, l'événement maximal de c'_p étant étiqueté par a. Cela n'implique pas que l'on ait $c_p \xrightarrow{a} c'_p$, ni que l'on ait $c \xrightarrow{a} c'_p$. Il se peut en effet qu'il y ait plusieurs actions (n'appartenant pas à Σ_p) intermédiaires entre c_p et c'_p, en plus de l'action a. Au contraire, dans la logique LTrL, l'assertion $c \models ⓐ\beta$ signifie que l'on peut effectuer un a à partir de c et obtenir une configuration c' qui satisfait β. L'opérateur ⓐ de LTrL fait donc avancer d'une configuration, alors que l'opérateur $ⓐ_p$ de TrPTL fait avancer d'une p-vue.*

Par exemple, sur la Figure 5.1, on considère une trace sur un alphabet distribué $(\Sigma_i)_{i=1,2}$ avec $\Sigma_1 = \{a, b\}$ et $\Sigma_2 = \{b, c\}$. Les lettres a et c sont donc indépendantes. La configuration acb (en pointillés) satisfait la formule de TrPTL

$$ⓐ_1 ⓒ_2 ⓐ_1 ⓒ_2 \dots ⓐ_1 ⓒ_2 \underline{tt},$$

quelque soit le nombre de facteurs $ⓐ_1 ⓒ_2$ de cette formule, alors que cette même configuration satisfait la formule de LTL

$$ⓐ ⓒ \underline{tt},$$

mais ne *satisfait* pas

$$ⓐ ⓒ ⓐ ⓒ \underline{tt}.$$

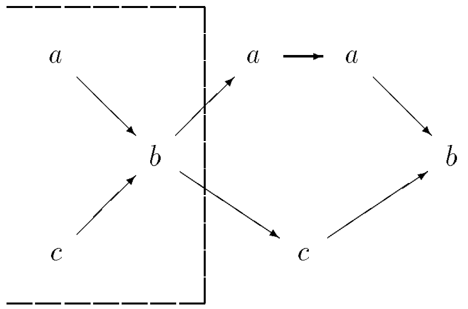

FIG. 5.1 –

5.3 Expressivité de LTrL

On dit qu'une trace u satisfait une formule ϕ de LTrL si $u, \emptyset \models \phi$. Le langage défini par une formule ϕ est $L(\phi) = \{u \in \mathbf{R}(\Sigma, D) \mid u, \emptyset \models \phi\}$. Un langage de traces $L \subseteq \mathbf{R}(\Sigma, D)$ est dit LTrL-définissable s'il existe une formule ϕ de LTrL telle que $L = L(\phi)$.

On rappelle (cf Chapitre 3) que la logique du premier ordre FO($\leq$) sur les traces est obtenue à partir de prédicats unaires du type $R_a(x)$ concernant les étiquettes des événements, des prédicats binaires du type $x \leq y$ concernant l'ordre (partiel) de ces événements, des connecteurs booléens $\neg$ et $\vee$, et des quantificateurs du type $\exists x$ et $\forall x$. Thiagarajan et Walukiewicz ont montré le théorème suivant :

Théorème 17 ([46]) *Soit $L \subseteq \mathbf{R}(\Sigma, D)$ un langage de traces réelles sur l'alphabet de dépendance (Σ, D). Alors les propositions suivantes sont équivalentes :*

(i) L est LTrL-définissable.

(ii) L est définissable dans la logique du premier ordre FO($\leq$).

L'équivalence décrite par le Théorème 17 comporte une implication facile $(i) \Rightarrow (ii)$ et une implication difficile $(ii) \Rightarrow (i)$. Pour montrer qu'un langage définissable dans LTrL l'est aussi dans FO($\leq$), il suffit de procéder par induction sur la structure de la formule ϕ de LTrL définissant ce langage. La seule difficulté consiste à décrire convenablement la notion de configuration dans FO($\leq$). Réciproquement, pour montrer qu'un langage définissable dans FO($\leq$) l'est aussi dans LTrL, Thiagarajan et Walukiewicz utilisent d'une part une décomposition particulière des traces réelles et d'autre part introduisent une notion de configuration *normale* qui peut être décrite par une formule de LTrL. Ils utilisent également l'équivalence entre la logique LTL sur les mots et la logique du premier ordre sur les mots, et montrent finalement qu'à toute formule α de LTL on peut faire correspondre une formule β de LTrL telle que les linéarisations *normales* des traces satisfaisant β soient exactement les mots satisfaisant la formule α. Leur démonstration fait appel à tous les opérateurs de LTrL, ainsi qu'aux formules de présent.

Dans ce chapitre, nous nous intéressons au problème de la *traduction* des formules de LTrL vers des automates reconnaissants le même langage. L'existence d'une telle traduction permet de résoudre certains problèmes de la vérification, comme le model-checking ou la satisfaisabilité, en raisonnant sur les automates plutôt que sur les formules. Notre résultat principal est une traduction *modulaire* (i.e. *inductive* sur la structure des formules) et s'énonce comme suit :

Théorème 18 *Soit ψ une formule de la logique temporelle LTrL, pour un alphabet de dépendance donné (Σ, D). On peut construire, de façon inductive sur la structure de ψ, un automate de Büchi $\mathcal{A}_\psi$ sur Σ^∞ qui accepte exactement les linéarisations des traces satisfaisant ψ.*

5.4 Méthode

5.4.1 Un exemple introductif

Construire un tel automate de façon inductive sur la structure de la formule ψ semble une méthode simple et naturelle, mais dans le cadre de LTrL cette approche se heurte à une petite difficulté, dûe à l'existence des *formules de présent* dans la logique. Considérons l'exemple suivant : soit $\psi = \underline{a}$. Alors le langage défini par ψ est vide, ce que l'on note $L(\psi) = \emptyset$, car la configuration vide n'a pas de lettres maximales. On a néanmoins $L(\textcircled{a}\underline{a}) = a\mathbf{R}(\Sigma, D)$, et pour $b \in \Sigma$ tel que $(a, b) \in D$, $L(\textcircled{b}\underline{a}) = \emptyset$. On voit sur cet exemple que la connaissance de $L(\psi)$ (ou celle d'un automate reconnaissant $L(\psi)$) ne suffit pas à déterminer le langage défini par une formule du type $@\psi$.

Ceci vient du fait que les formules du type $\underline{a}$ de LTrL ne sont pas strictement orientées vers le futur. Afin de surmonter cette difficulté, on est amené à prendre également en compte le *contexte de présent* des traces. De façon plus formelle, soit $u \in \mathbf{R}(\Sigma, D)$ une trace réelle, et soient $v, v' \in \mathbf{M}(\Sigma, D)$ deux traces finies. Considérons les traces réelles vu et $v'u$. On identifie les préfixes finis d'une trace avec ses configurations. Alors il existe (a priori) une formule ψ de LTrL telles que $vu, v \models \psi$ mais $v'u, v' \not\models \psi$, comme nous l'avons vu sur l'exemple précédent. Néanmoins, si $\max(v) = \max(v')$, alors $vu, v \models \psi$ ssi $v'u, v' \models \psi$, où l'on rappelle que $\max(v)$ désigne l'ensemble des lettres maximales du préfixe v.

Pour déterminer la validité d'une formule sur une trace t et en un préfixe v il suffit donc de connaître :

- le suffixe w de t tel que $t = vw$;
- les lettres maximales de v.

Cette remarque est formalisée par le Lemme suivant :

Lemme 4 *Soit ψ une formule de LTrL, et soient $u \in \mathbf{R}(\Sigma, D)$ et $v, v' \in \mathbf{M}(\Sigma, D)$. On considère les traces réelles vu et $v'u$. Si $\max(v) = \max(v')$, alors*

$$vu, v \models \psi \text{ ssi } v'u, v' \models \psi.$$

Cette remarque nous amène à définir la notion d'*ensemble compatible*, ce qui est l'objet du paragraphe suivant.

5.4.2 Ensembles compatibles

Comme deux lettres maximales d'une configuration sont nécessairement indépendantes, nous introduisons la notion d'ensemble *compatible* suivante : on dit qu'un ensemble $B \subseteq \Sigma$ de lettres est *compatible* (avec I) si $B \times B \subseteq I$. Un ensemble compatible est donc une clique du graphe (Σ, I). Ainsi, tout ensemble de lettres maximales d'un préfixe fini est compatible, et tout ensemble compatible peut être l'ensemble des lettres maximales d'un préfixe fini.

Pour toute formule ψ de LTrL nous allons contruire inductivement une famille finie d'automates $(\mathcal{A}_\psi^B)_B$ pour tous les ensembles B compatibles. Rappelons que φ désigne l'application canonique de Σ^∞ dans $\mathbf{R}(\Sigma, D)$. L'automate $\mathcal{A}_\psi^B$ reconnaîtra le langage de mots $\varphi^{-1}(L_B(\psi))$, où

$$L_B(\psi) = \{u \in \mathbf{R}(\Sigma, D) \mid \forall v \in \mathbf{M}(\Sigma, D), \max(v) = B \Rightarrow vu, c_v \models \psi\}.$$

Dans cette famille, l'automate qui nous intéresse est $\mathcal{A}_\psi^\emptyset$, puisqu'il reconnaît le langage $\varphi^{-1}(L_\emptyset(\psi))$, et que $L_\emptyset(\psi) = L(\psi)$ est le langage de traces défini par ψ.

La notion d'ensemble compatible nous amène à définir une sémantique plus appropriée pour LTrL. En effet, nous avons vu que la satisfaction d'une formule pour un modèle u, v ne dépend que de $\max(v)$ et du suffixe correspondant à v dans u, c'est-à-dire à la trace w telle que $u = vw$.

Pour cela, on définit inductivement la fonction $\odot$ de $\Sigma \times \mathbf{M}(\Sigma, D)$ dans Σ par :

- $B \odot \epsilon = B$, pour tout $a \in \Sigma$.
- $B \odot (va) = ((B \odot v) \cap I(a)) \cup \{a\}$, pour tout $v \in \mathbf{M}(\Sigma, D)$.

On vérifie aisément que cette fonction est définie sans ambiguité sur $\Sigma \times \mathbf{M}(\Sigma, D)$, et qu'elle transforme un ensemble compatible en un autre ensemble compatible. Plus précisemment, on a :

Lemme 5 *Soient* $v, v' \in \mathbf{M}(\Sigma, D)$*. Si* $\max(v) = B$*, alors* $\max(vv') = B \odot v'$.

On peut maintenant définir pour toute trace $u \in \mathbf{R}(\Sigma, D)$ et tout ensemble compatible B, la sémantique suivante des formules de LTrL :

- $u, B \models \underline{tt}$.
- $u, B \models \underline{a}$ ssi $a \in B$.
- $u, B \models \neg\alpha$ ssi $u, B \not\models \alpha$.
- $u, B \models \alpha \vee \beta$ ssi $u, B \models \alpha$ ou $u, B \models \beta$.
- $u, B \models @\alpha$ ssi il existe $v \in \mathbf{R}(\Sigma, D)$ telle que $u = av$ et $v, B \odot a \models \alpha$.
- $u, B \models \alpha\ \mathcal{U}\ \beta$ ssi il existe $v \in \mathbf{M}(\Sigma, D)$ et $v' \in \mathbf{R}(\Sigma, D)$ telles que $u = vv'$ et $v', B \odot v \models \beta$ et pour toute décomposition $v = v_1 v_2$, si $v_2 \neq \epsilon$ alors $v_2 v', B \odot v_1 \models \alpha$.

Le lien avec la sémantique donnée précédemment est le suivant (on rappelle qu'on identifie les préfixes finis d'une trace avec ses configurations) :

Lemme 6 *Soit ψ une formule de LTrL. Soit $u \in \mathbf{R}(\Sigma, D)$ et soit $v \in C_u$. Posons $u = vv'$, avec $v' \in \mathbf{R}(\Sigma, D)$. Alors*

$$u, v \models \psi \textit{ ssi } v', B \models \psi,$$

où $B = \max(v)$.

Dans la suite, on préfèrera pour des raisons de commodité utiliser cette nouvelle sémantique de LTrL. En particulier, le langage $L_B(\psi)$ se définit plus naturellement :

$$L_B(\psi) = \{u \in \mathbf{R}(\Sigma, D) \mid u, B \models \psi\}.$$

5.5 Outils

Rappelons ici que le résultat principal de ce chapitre est la construction modulaire d'automates de Büchi à partir de formules de LTrL. Bien qu'il existe des classes d'automates qui reconnaissent directement des traces (en tant que graphes de dépendance), il semble plus simple dans notre cadre de chercher à construire des automates sur les mots, qui reconnaissent des linéarisations de traces, c'est-à-dire des langages de mots I-clos. Plus formellement, un langage de mots $L \subseteq \Sigma^\infty$ est dit I-clos s'il est clos par la projection canonique φ, i.e. si $L = \varphi^{-1}\varphi(L)$. Un langage est donc I-clos si c'est l'image inverse par φ d'un langage de traces.

5.5.1 Automates de Büchi sur Σ^∞

Les automates que nous allons construire sont *automates de Büchi* (non-déterministes) *généralisés* sur Σ^∞.

Définition

Un automate de Büchi *généralisé* sur Σ^∞ est un quintuplet $\mathcal{A} = (Q, \rightarrow, S, F, R)$ tel que :

- Q est un ensemble *fini*, dit ensemble des états de l'automate;
- $\rightarrow \subseteq Q \times \Sigma \times Q$ est la *relation de transition*;

- $S \subseteq Q$ est l'ensemble des états *initiaux* de l'automate;
- $F \subseteq Q$ est l'ensemble des états *finaux*;
- $R \subseteq 2^Q$ est un ensemble fini de sous-ensembles d'états *répétés*.

Remarquons que cette définition diffère de celle classique d'un automate de Büchi par la nature de la condition d'acceptation, comme nous le verrons en détail dans les paragraphes ci-dessous.

Une transition $(q, a, q') \in \rightarrow$ est souvent notée $q \xrightarrow{a} q'$. De plus, la fermeture réflexive et transitive de $\rightarrow$ est notée $\rightarrow^*$.

Exécutions

Dans notre contexte, un automate de Büchi généralisé permet d'avoir des exécutions aussi bien finies qu'infinies. Formellement, une exécution (finie) de l'automate $\mathcal{A} = (Q, \rightarrow, S, F, R)$ sur un mot fini $w = a_1 a_2 \ldots a_n$ est une suite finie $\tau = (q_0, a_1, q_1, a_2, \ldots, a_n, q_n)$ telle que :

- $q_0 \in S$ et pour tout $i > 0$, $q_i \in Q$.
- Pour tout i, $0 \leq i \leq n-1$, on ait $q_i \xrightarrow{a_{i+1}} q_{i+1}$.

Dans ce cas, on note $\mathrm{fin}(\tau)$ le dernier état de τ, c'est-à-dire q_n.

Si $w = a_1 a_2 a_3 \ldots$ est un mot infini, une exécution de $\mathcal{A}$ sur w est une suite infinie $\tau = (q_0, a_1, q_1, a_2, \ldots)$ telle que :

- $q_0 \in S$ et pour tout $i > 0$, $q_i \in Q$.
- Pour tout $i \geq 0$, $q_i \xrightarrow{a_{i+1}} q_{i+1}$.

Dans ce cas, on note $\inf(\tau)$ l'ensemble des états infiniment répétés dans τ, c'est-à-dire :

$$\inf(\tau) = \{q \in Q \mid \mathrm{Card}(\{i \geq 0 \mid q_i = q\}) = \omega\}.$$

Conditions d'acceptation

Si w est un mot fini, on dit que l'automate $\mathcal{A}$ *accepte* w (ou *reconnaît* w) s'il *existe* une exécution τ de $\mathcal{A}$ sur w telle que $\mathrm{fin}(\tau) \in F$.

Si w est un mot infini, on dit que l'automate $\mathcal{A}$ *accepte* w (ou *reconnaît* w) s'il *existe* une exécution τ de $\mathcal{A}$ sur w telle que, pour tout $r \in R$,

$\inf(\tau) \cap r \neq \emptyset$. Une telle condition est dite *condition d'acceptation de Büchi généralisée*. Il s'agit en fait d'une *conjonction* de conditions d'acceptation de Büchi classiques.

On note $L(\mathcal{A})$ le langage reconnu par $\mathcal{A}$, i.e. l'ensemble des mots (finis ou infinis) reconnus par l'automate $\mathcal{A}$.

Lemme 7 *Les automates de Büchi "classiques" et les automates de Büchi* généralisés *ont le même pouvoir d'expression (i.e. reconnaissent la même classe de langages).*

Preuve. Nous allons montrer que l'utilisation d'une conjonction finie dans les conditions d'acceptation d'un mot infini n'augmente pas la puissance des automates de Büchi. En effet, tout automate de Büchi peut être vu comme un automate de Büchi généralisé. Réciproquement, si $\mathcal{A}$ est un automate de Büchi généralisé dont la table d'acceptation est R, alors un mot infini w est reconnu par $\mathcal{A}$ ssi il existe une exécution τ telle que pour tout $r \in R$, $\inf(\tau) \cap r \neq \emptyset$. Soit R' l'ensemble des parties r' de Q telles que pour tout $r \in R$, $r \cap r' \neq \emptyset$. Alors w est accepté ssi il existe $r' \in R'$ tel que $r' \subseteq \inf(\tau)$. On définit $K = \{k \subseteq Q \mid \exists r' \in R', r' \subseteq k\}$. La condition d'acceptation devient alors : il existe $k \in K$ tel que $\inf(\tau) = k$. Il s'agit donc d'une condition d'acceptation *à la Muller* [34], et on sait que les automates de Muller reconnaissent les mêmes langages de mots infinis que les automates de Büchi classiques. □

Nous nous permettons donc d'utiliser cette condition d'acceptation généralisée, qui permettra par la suite plus de simplicité dans l'écriture des automates. De plus, on omettra la plupart du temps de préciser qu'il s'agit d'automates de Büchi *généralisés*.

5.5.2 Automates alphabétiques

Dans les constructions que nous allons effectuer, nous aurons parfois besoin que les automates déjà construits soient *alphabétiques*, c'est-à-dire qu'ils contiennent, en plus de leurs états, des indications concernant l'alphabet des mots utilisés pour atteindre ces états.

Formellement, un automate de Büchi $\mathcal{A} = (Q, \rightarrow, S, F, R)$ sur Σ^∞ est *alphabétique* si pour tout état $q \in Q$ il existe une partie B de Σ telle que :

$$\forall u \in \Sigma^\infty, \forall q_0 \in S, (q_0 \rightarrow^* q) \Rightarrow (\text{alph}(u) = B)$$

où $\mathrm{alph}(u)$ est l'ensemble des lettres apparaissant dans u. Si c'est le cas, alors on peut noter $\mathrm{alph}(q) = B$.

La proposition suivante montre que tout automate de Büchi peut être transformé en un automate de Büchi alphabétique équivalent (i.e. reconnaissant le même langage).

Proposition 11 *Soit $\mathcal{A}$ un automate de Büchi sur Σ^∞. Alors on peut construire un automate de Büchi $\mathcal{A}'$ sur Σ^∞ tel que :*

- *$\mathcal{A}'$ soit alphabétique;*
- *$L(\mathcal{A}') = L(\mathcal{A}')$.*

Cette proposition est classique, et nous rappelons brièvement sa preuve.

Preuve. Soit $\mathcal{A} = (Q, \rightarrow, S, F, R)$. On définit $\mathcal{A}' = (Q', \rightarrow', S', F', R')$ de la façon suivante :

- $Q' = Q \times 2^\Sigma$;
- $(q, B) \xrightarrow{a}{}' (q', B')$ ssi $q \xrightarrow{a} q'$ et $B' = B \cup \{a\}$;
- $S' = \{(q, \emptyset) \mid q \in S\}$;
- $F' = \{(q, B) \in Q' \mid q \in F \text{ et } B \subseteq \Sigma\}$;
- $R = \{r \times B \mid r \in R \text{ et } B \subseteq \Sigma\}$.

Alors un mot w de Σ^∞ est accepté par $\mathcal{A}$ ssi il est accepté par $\mathcal{A}'$, donc $L(\mathcal{A}) = L(\mathcal{A}')$. De plus, $\mathcal{A}'$ est alphabétique, car on montre par une induction immédiate que si un état (q, B) est atteignable à partir d'un état initial $(q_0, \emptyset)$ à travers un mot w, alors $\mathrm{alph}(w) = B$. □

Grâce à ces outils (automates de Büchi généralisés, automates alphabétiques), nous allons maintenant pouvoir aborder le coeur de nos constructions.

5.6 Constructions modulaires

La modularité de nos constructions réside dans le fait qu'elles s'effectuent de façon inductive par rapport à la structure de la formule étudiée. Il suffit donc d'expliciter ces constructions pour la formule la plus simple, i.e. la formule constante $\underline{tt}$, et ensuite de préciser comment effectuer des constructions pour une formule ψ quelconque en fonction de constructions déjà effectuées pour les sous-formules de ψ. Les différents paragraphes qui suivent correspondent donc aux différents cas de structure de ψ. Notons que la difficulté majeure de ces constructions réside dans le cas où ψ est de la forme $\beta\,\mathcal{U}\,\gamma$.

5.6.1 La formule constante

On suppose que $\psi = \underline{tt}$. Alors, pour tout ensemble compatible B, $L_B(\psi) = L_B(\underline{tt}) = \mathbf{R}(\Sigma, D)$, et par conséquent $\varphi^{-1}(L_B(\psi)) = \Sigma^\infty$. Tout langage de mots doit alors être accepté par l'automate $\mathcal{A}^B_{\underline{tt}}$. La construction d'un tel automate est donc triviale; on peut prendre par exemple l'automate $\mathcal{A}^B_{\underline{tt}} = (Q, \rightarrow, S, F, R)$ suivant :

- $Q = \{0\}$;
- $0 \xrightarrow{a} 0$ pour toute lettre $a \in \Sigma$;
- $S = F = Q$;
- $R = \{Q\}$.

5.6.2 Les formules de présent

Supposons maintenant que $\psi = \underline{a}$ pour une lettre $a \in \Sigma$. Selon la nature de l'ensemble compatible B, nous devons effectuer deux constructions distinctes. En effet, si $a \in B$, alors toute trace u est dans $L_B(\underline{a})$, car le "contexte de présent" indique que a est maximale. On a donc $L_B(\underline{a}) = \mathbf{R}(\Sigma, D)$, et il suffit de définir $\mathcal{A}^B_{\underline{a}} = \mathcal{A}^B_{\underline{tt}}$.

Par contre, si $a \notin B$, cela signifie que a n'est pas une lettre maximale dans le contexte de présent, donc $L_B(\underline{a}) = \emptyset$. On définit alors $\mathcal{A}^B_{\underline{a}} = \mathcal{A}_{\neg\underline{tt}}$, où $\mathcal{A}^B_{\neg\underline{tt}}$ étant similaire à $\mathcal{A}^B_{\underline{tt}}$ si ce n'est que l'on pose cette fois $F = \emptyset$ et $R = \emptyset$, de façon à ce qu'aucune trace ne soit acceptée par l'automate.

5.6.3 La disjonction

Supposons que $\psi = \alpha \vee \beta$, et que l'on dispose, pour tout ensemble compatible B des automates $\mathcal{A}_\alpha^B$ et $\mathcal{A}_\beta^B$. La sémantique du connecteur booléen $\vee$ implique que $L_B(\psi) = L_B(\alpha) \cup L_B(\beta)$. On peut supposer, sans perte de généralité, que les automates $\mathcal{A}_\alpha^B$ et $\mathcal{A}_\beta^B$ ont des ensembles d'états *disjoints*. Notons $\mathcal{A}_\alpha^B = (Q_1, \rightarrow_1, S_1, F_1, R_1)$ et $\mathcal{A}_\beta^B = (Q_2, \rightarrow_2, S_2, F_2, R_2)$. On définit alors $\mathcal{A}_\psi^B = (Q, \rightarrow, S, F, R)$ comme suit :

- $Q = Q_1 \cup Q_2$ (union disjointe);
- $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ (union des relations);
- $S = S_1 \cup S_2$;
- $F = F_1 \cup F_2$;
- $R = \{(r_1 \cup Q_2) \mid r_1 \in R_1\} \cup \{(r_2 \cup Q_1) \mid r_2 \in R_2\}$.

L'exécution τ sur un mot w se déroule alors soit exclusivement dans Q_1, soit exclusivement dans Q_2. Si par exemple elle se déroule dans Q_1, alors $\inf(\tau) \subseteq Q_1$, donc une condition du type $\inf(\tau) \cap (r_2 \cup Q_1) \neq \emptyset$ est toujours vérifiée, indépendament de $r_2 \in R_2$. De plus, une condition du type $\inf(\tau) \cap (r_1 \cup Q_2) \neq \emptyset$, pour un $r_1 \in R_1$, se résume à $\inf(\tau) \cap r_1 \neq \emptyset$. Il est alors immédiat qu'un mot w est accepté par $\mathcal{A}_\psi^B$ s'il est accepté soit par $\mathcal{A}_\alpha^B$, soit par $\mathcal{A}_\beta^B$. Comme l'union des langages commute avec φ^{-1}, l'automate $\mathcal{A}_\psi^B$ reconnaît bien le langage $\varphi^{-1}(L_B(\alpha)) \cup \varphi^{-1}(L_B(\beta)) = \varphi^{-1}(L_B(\psi))$.

5.6.4 La négation

On suppose que $\psi = \neg\alpha$. Pour tout ensemble compatible B, l'automate $\mathcal{A}_\psi^B$, qui doit reconnaître le langage $\varphi^{-1}(\mathbf{R}(\Sigma, D) \backslash L_B(\alpha)) = \Sigma^\infty \backslash \varphi^{-1}(L_B(\alpha))$, peut être obtenu par complémentation de l'automate $\mathcal{A}_\alpha^B$, par exemple par la méthode classique de Safra [42]. Néanmoins, la méthode de Safra fonctionnant sur des automates de Muller, il convient au préalable de transformer $\mathcal{A}_\alpha^B$ en un automate de Muller, comme indiqué dans la preuve du Lemme 7. Une fois l'automate complémentaire obtenu, il convient également de le transformer en un automate de Büchi généralisé, ce qui se fait également comme indiqué dans la preuve du Lemme 7. Notons que cette complémentation entraine une explosion exponentielle de la taille de l'automate, c'est-à-dire que l'automate obtenu $\mathcal{A}_\psi^B$ est exponentiellement plus grand que l'automate d'origine $\mathcal{A}_\alpha^B$.

5.6.5 Les opérateurs de type *next*

On suppose à présent que $\psi = @\alpha$, pour un certain $a \in \Sigma$. Pour une trace u et un contexte de présent B, on sait que $u, B \models @\alpha$ ssi il existe v telle que $u = av$, et $v, B \odot a \models \alpha$. Du point du vue des langages de traces, cela s'écrit $L_B(@\alpha) = aL_C(\alpha)$, avec $C = B \odot a$.

Grâce à cette remarque, nous allons construire l'automate $\mathcal{A}_\psi^B$ à partir de l'automate $\mathcal{A}_\alpha^C$. Il s'agit de construire une sorte de produit de $\mathcal{A}_\alpha^C$ avec un automate dont le seul but est de vérifier la présence de la lettre a comme lettre minimale. Ce second automate "attend" donc une lettre a. Tant qu'on lit des lettres indépendantes de a, on continue à attendre, tout en les faisant agir dans $\mathcal{A}_\alpha^C$. Si on lit un b dépendant de a, on va vers un état puits, car cela signifie que a n'est pas minimale dans la trace lue. Enfin, si on lit a, on passe dans un état d'acceptation, où il reste à faire agir le reste du mot dans l'automate $\mathcal{A}_\alpha^C$.

Notons $\mathcal{A}_\alpha^C = (Q_1, \to_1, S_1, F_1, R_1)$. On définit alors $\mathcal{A}_\psi^B = (Q, \to, S, F, R)$ comme suit :

- $Q = Q_1 \times \{wait, ok, fail\}$;
- $S = S_1 \times \{wait\}$;
- $F = F_1 \times \{ok\}$;
- $R = \{(r \times \{ok\}) \mid r \in R_1\}$;

et on définit $\to$ de la façon suivante :

- $(p, wait) \overset{b \in I(a)}{\longrightarrow} (p', wait)$ ssi $p \overset{b}{\to}_1 p'$;
- $(p, wait) \overset{b \in D(a) \setminus \{a\}}{\longrightarrow} (p', fail)$ ssi $p \overset{b}{\to}_1 p'$;
- $(p, wait) \overset{a}{\longrightarrow} (p', ok)$ ssi $p \overset{a}{\to}_1 p'$;
- $(p, ok) \overset{b}{\to} (p', ok)$ ssi $p \overset{b}{\to}_1 p'$;
- $(p, fail) \overset{b}{\to} (p', fail)$ ssi $p \overset{b}{\to}_1 p'$.

Cette construction garantit que $L(\mathcal{A}_\psi^B) = \varphi^{-1}(a\varphi(L(\mathcal{A}_\alpha^C)))$, qui est la propriété désirée.

5.6.6 L'opérateur *until*

Nous abordons maintenant la construction dans laquelle réside la difficulté principale de ce chapitre. Commençons par rappeler le principe de la vérification modulaire d'une formule du type $\beta\,\mathcal{U}\,\gamma$ dans le cas des *mots* : si un mot u satisfait $\beta\ \mathcal{U}\ \gamma$, alors il existe des mots v et w tels que $u = vw$, w satisfait γ et pour tout suffixe non vide v' de v, $v'w$ satisfait β. Le mot v étant nécessairement fini, on pose $v = v_1v_2 \dots v_k$. La vérification de $\beta\ \mathcal{U}\ \gamma$ s'effectue alors naturellement en deux phases successives :

1. lecture des lettres v_i de v, et initialisation d'une vérification de β pour chacun des suffixes $v_iv_{i+1} \dots v_kw$.

2. lecture des lettres de w, vérification de γ pour le suffixe w, et poursuite des vérifications de β commencées à l'étape précédente.

Remarquons que comme on ne connaît pas a priori la décomposition vw de u qui convient, le passage de la première phase à la deuxième s'effectue de façon non-déterministe. Le principe de modularité de cette méthode implique que, quand on cherche à construire un automate reconnaissant le même langage que $\beta\ \mathcal{U}\ \gamma$, on connaît déjà (i.e. on a déjà construit) des automates $\mathcal{A}_\beta$ et $\mathcal{A}_\gamma$ reconnaissant respectivement les langages définis par les formules β et γ. Ainsi, lorsqu'on parle d'initialiser une vérification de β lors de la lecture d'une lettre a, il s'agit en fait d'appliquer l'action a dans l'automate $\mathcal{A}_\beta$, en partant d'un état initial. De même, lorsqu'on parle de continuer une vérification d'une formule lors de la lecture d'une lettre a, il s'agit de faire exécuter l'action a dans l'automate correspondant, à partir de l'état courant pour cette vérification. Comme on cherche à construire un automate *fini*, on rencontre une difficulté liée au fait que l'on ne peut a priori pas borner le nombre d'exécutions dans $\mathcal{A}_\beta$ à simuler. Afin de contourner ce problème, chaque exécution en cours dans $\mathcal{A}_\beta$ est représentée par son état actuel dans cet automate, et on garde en mémoire dans l'automate $\mathcal{A}_{\beta\mathcal{U}\gamma}$ un ensemble P d'états distincts de $\mathcal{A}_\beta$, qui sont évidemment en nombre fini. Lorsque deux exécutions arrivent dans le même état, on ne garde ainsi qu'une seule copie de cet état. Comme ces exécutions vont subir l'action du même mot dans le futur, si l'une peut être acceptante (de façon non-déterministe), alors l'autre peut l'être aussi, et il est donc légitime de n'en garder qu'un seul représentant. Nous verrons par la suite que le cas des traces est légèrement différent à cet égard.

Dans le cas des *traces*, nous allons appliquer un principe identique à celui explicité dans le cas des mots, mais en considérant des I-suffixes au lieu des suffixes classiques. Rappelons qu'un mot w est un I-suffixe d'un mot u s'il existe un mot v tel que $\varphi(vw) = \varphi(u)$. Explicitons ce principe sur un exemple. On suppose que $\Sigma = \{a, b, c\}$, et que seules les lettres a et c sont indépendantes. Considérons la trace suivante :

$$t = [acbaacb].$$

Supposons que t satisfasse $\beta\, \mathcal{U}\, \gamma$ pour la décomposition $t = [acba][acb]$. On pose $x = acba$ et $y = acb$. Lorsqu'on lit une linéarisation w de t, on ne lit pas nécessairement toutes les lettres du préfixe x en premier, puis toutes les lettres du suffixe y :

$$w = \mathbf{acbca}ab$$

est une linéarisation de t dans laquelle les lettres du préfixe sont en gras, et les lettres du suffixe en caractères normaux. Lors de la première phase, on va donc être amené a "sauter" des lettres de y, tant que toutes les lettres de x n'ont pas été lues. Néanmoins, il convient de faire agir les lettres que l'on saute ainsi sur la vérification de la formule γ. Il s'agit ici d'une difficulté inhérente aux traces, c'est-à-dire qui n'apparaît pas dans le cas des mots.

Une autre difficulté inhérente aux traces est que l'on peut être amené à lancer plusieurs vérifications de β lors de la lecture d'une seule lettre de x. Reprenons l'étude de notre exemple. Lorsqu'on lit la première lettre de w, c'est-à-dire a, on doit lancer une vérification de β correspondante au I-suffixe $acbaab$. Pour cela, il suffit de faire agir a dans un automate "reconnaissant" β. Ensuite, lorsqu'on lit la lettre c, on doit lancer la vérification correspondante au I-suffixe $cbaacb$ (i.e. faire agir c dans une nouvelle exécution de l'automate de β), mais aussi celle correspondant au I-suffixe $abaacb$, puisque a et c commutent (i.e. conserver une copie inchangée de la vérification commencée avec a), et enfin continuer les vérifications lancées précédemment (i.e. faire agir c sur l'exécution commencée avec a). De façon plus générale, lorsqu'on lit une nouvelle lettre l de x, on doit effectuer les mises-à-jour suivantes :

1. faire agir l sur les vérifications de β déjà lancées.
2. faire agir l sur une nouvelle vérification de β (il faut tenir compte des éventuelles lettres de y déjà lues).
3. conserver les vérifications de β lancées auparavant pour lesquelles l'alphabet utilisé jusqu'alors commute avec l.

Remarquons que dans le cas des mots seules les deux premières mise-à-jours sont nécessaires.

Illustrons ces différents points sur l'exemple précedemment introduit. La tableau ci-dessous montre les vérifications de β et de γ en cours à chaque étape (i.e. à chaque lecture d'une nouvelle lettre du mot w). La première colonne indique la lettre de w que l'on est en train de lire, la seconde indique les vérifications de β en cours, que l'on a indéxées par 1, 2 ou 3 suivant le type d'action dont elles proviennent, cet indice correspondant à l'énumération ci-dessus; enfin, la troisième colonne indique l'avancement de la vérification de γ en cours. Quand on lit une lettre de y, seules des mise-à-jours de type 1 interviennent dans les vérifications de β. Pour faciliter la lecture, on a écrit en gras les lettres de x.

lettre	β-vérifications	γ-vérifications
$\mathbf{a}$	$\mathbf{a}_2$	
$\mathbf{c}$	$\mathbf{ac}_1, \mathbf{c}_2, \mathbf{a}_3$	
$\mathbf{b}$	$\mathbf{acb}_1, \mathbf{cb}_1, \mathbf{ab}_1, \mathbf{b}_2$	
c	$\mathbf{acb}c_1, \mathbf{cb}c_1, \mathbf{ab}c_1, \mathbf{b}c_1$	c
$\mathbf{a}$	$\mathbf{acb}c\mathbf{a}_1, \mathbf{cb}c\mathbf{a}_1, \mathbf{ab}c\mathbf{a}_1, \mathbf{b}c\mathbf{a}_1, c\mathbf{a}_2$	c
a	$\mathbf{acb}c\mathbf{a}a_1, \mathbf{cb}c\mathbf{a}a_1, \mathbf{ab}c\mathbf{a}a_1, \mathbf{b}c\mathbf{a}a_1, c\mathbf{a}a_1$	ca
b	$\mathbf{acb}c\mathbf{a}ab_1, \mathbf{cb}c\mathbf{a}ab_1, \mathbf{ab}c\mathbf{a}ab_1, \mathbf{b}c\mathbf{a}ab_1, c\mathbf{a}ab_1$	cab

On voit sur cet exemple que l'on a bien effectué une β-vérification pour chacun des I-suffixes non-vides de x.

Par ailleurs, on va ici être amené à garder en mémoire, pour chaque vérificationde β, l'ensemble des états accessibles par la lecture du préfixe déjà lu, c'est-à-dire que l'on va simuler une déterminisation de l'automate lié à β. En effet, il n'est plus question ici de fusionner deux vérifications de β conduisant à un état commun, car les mots restant à lire peuvent différer d'une vérification à l'autre, puisqu'on est parfois amené à sauter certaines lettres dans une vérification mais pas dans l'autre. On va donc garder en mémoire un ensemble P d'ensemble d'états, et on ne fusionnera deux vérifications que lorsque les ensembles d'états accessibles dans l'une et dans l'autre coïncident.

Enfin, une dernière difficulté vient de la présence des formules de présent dans la logique temporelle LTrL. En effet, lorsqu'on lance une nouvelle exécution, il s'agit de l'effectuer dans le "bon" contexte de présent. Il nous faut donc mettre à jour, à mesure de la lecture du mot w, les différents contextes de présents liés aux I-préfixes de w. Remarquons que l'on est ici amené à “deviner”, de manière non-déterministe, le contexte de présent dans lequel on lance une vérification, car il se peut que l'on ait encore à lire des lettres qui vont modifier ce contexte. Dans notre exemple, lorsqu'on lance la vérification de γ à la lecture du second c, on est amené à deviner un contexte de présent dans lequel le a qui suit (donc que l'on n'a pas encore lu) intervient. Il s'agira donc par la suite de vérifier a posteriori que nos suppositions étaient fondées.

Nous abordons maintenant la contruction de l'automate. Supposons un contexte de présent B fixé, et cherchons à construire l'automate correspondant à la formule $\psi = \beta\ \mathcal{U}\ \gamma$ dans le contexte B, c'est-à-dire reconnaissant le langage $\varphi^{-1}(L_B(\beta\ \mathcal{U}\ \gamma))$. Supposons que l'on soit en train de lire une linéarisation m d'une trace u. Si u satisfait $\beta\ \mathcal{U}\ \gamma$, alors u s'écrit $u = vw$, avec $w \models \gamma$ et pour tout suffixe non vide v' de v, $v'w \models \beta$. Il existe alors des sous-mots x et y de m tels que

$$\begin{array}{rcl} m & \in & x \sqcup\!\sqcup_I y \\ \varphi(x) & = & v \\ \varphi(y) & = & w. \end{array}$$

où l'on rappelle que $x \sqcup\!\sqcup_I y$ est l'ensemble des mots dont l'image par φ est $\varphi(x)\varphi(y)$.

Comme nous l'avons vu, cet automate va agir en deux phases distinctes. La première phase, qui va être effectuée par un premier automate $\mathcal{B}$, persiste tant que toutes les lettres de x n'ont pas été lues dans w. Lors de cette phase, on est amené à effectuer deux types de transitions :

- Si on lit une lettre de x, on lance des nouvelles vérifications de β et on continue les vérifications de β déjà lancées. Une telle transition est dite β-transition.

- Si on lit une lettre de y, on commence (si c'est la première lettre de y) ou on continue (dans le cas contraire) une vérification de γ, tout en mettant à jour les vérifications de β existantes. Une telle transition est dite γ-transition.

Remarquons que, puisqu'on ne connaît pas a priori la décomposition (x, y) de m, le choix d'effectuer une β-transition ou une γ-transition est non-déterministe. Les β-transitions sont néanmoins soumises à la garde suivante : la lettre que l'on lit doit commuter avec toutes les lettres lues lors des γ-transitions déjà effectuées. Ceci permet de s'assurer que l'on a bien $m \in x \sqcup\!\sqcup_I y$.

De façon non-déterministe, on peut décider que toutes les lettres de x ont été lues, et passer, *via* une ϵ-transition, vers un automate $\mathcal{C}$ qui ne fera alors que continuer toutes les vérifications commencées auparavant, et qui permettra d'évaluer la validité de ces vérifications grâce à des conditions d'acceptation adéquates.

Automate $\mathcal{B}$

Comme on va avoir à effectuer des vérifications de β pour différents contextes de présent, on notera, pour des raisons de commodité, $\mathcal{A}_1$ l'union *disjointe* des automates $\mathcal{A}_\beta^C$ pour tous les ensembles compatibles C. De même, on notera $\mathcal{A}_2$ l'union disjointe des automates $\mathcal{A}_\gamma^C$ pour tous les ensembles C compatibles. Pour $i = 1, 2$, on note Q_i l'ensemble des états de $\mathcal{A}_i$. On suppose que $Q_1 \cap Q_2 = \emptyset$.

Un état de $\mathcal{B}$ est de la forme (P, X, E, Y), où :

- P est un ensemble de couple (Z, G), chaque couple correspondant à une vérification de β liée à un ou plusieurs I-suffixes de v. Dans un couple (Z, G), Z est un ensemble d'états accessibles de $\mathcal{A}_1$, i.e. un sous-ensemble de Q_1, et G est un ensemble compatible représentant le contexte de présent dans lequel cette vérification a été lancée. Il n'y a donc qu'un nombre fini de couples (Z, G) possibles.

- $X \subseteq Q_1$ représente tous les états de $\mathcal{A}_1$ accessibles après lecture du préfixe actuel de y. Cet ensemble permet de mettre à jour correctement l'ensemble P, car lorsqu'on lance une nouvelle vérification pour β on doit tenir compte de l'action des lettres de y déjà lues.

- $E \subseteq \Sigma$ permet de maintenir à jour le contexte de présent lié à la vérification de γ.

- $Y \subseteq Q_2$ représente la vérification en cours de γ. Notons qu'on effectue une déterminisation de $\mathcal{A}_2$, car on va garder en mémoire tous les états de Q_2 accessibles après lecture du préfixe actuel de y, pour des raisons de commodité (il ne s'agit pas, contrairement au cas des couples (Z, G), d'une nécessité).

L'ensemble des états de $\mathcal{A}_\psi^B$ est donc

$$Q_{\mathcal{B}} = \mathcal{P}(2^{Q_1} \times 2^{\Sigma}) \times 2^{Q_1} \times 2^{\Sigma} \times 2^{Q_2}.$$

L'unique état initial de $\mathcal{B}$ est $(\emptyset, S_1, B, S_2)$, où S_i $(i = 1, 2)$ est l'ensemble des états initiaux de l'automate $\mathcal{A}_i$.

Il nous faut maintenant préciser la nature des γ-transitions et des β-transitions, ainsi que celle de l'ϵ-transition qui permettra la passage de l'automate $\mathcal{B}$ vers l'automate $\mathcal{C}$, c'est-à-dire le changement de phase. Pour cela, on définit d'abord la notation suivante : pour tout sous-ensemble W de Q_i (où $i = 1, 2$), et pour toute lettre a de Σ, on pose

$$W \cdot a = \{q' \in Q_i \mid \exists q \in W, q \xrightarrow{a}_i q'\}.$$

L'ensemble $W \cdot a$ est donc l'ensemble des états de Q_i accessibles à partir d'un état de W après lecture de a dans l'automate $\mathcal{A}_i$.

γ-transitions : Une γ-transition est toujours possible (i.e il n'y a pas de garde), et est de la forme :

$$(P, X, E, Y) \xrightarrow{a} (P \bullet a, X \cdot a, E, Y \cdot a)$$

où

$$P \bullet a = \{(Z \cdot a, G) \mid (Z, G) \in P\}.$$

Les ensembles X et Y sont donc mis à jour de façon naturelle. Pour l'ensemble P, on doit faire agir a sur chacun des Z, mais sans modifier les contextes G associés. L'ensemble E, lui, n'est pas modifié.

β-transitions : Une β-transition étiquetée par $a \in \Sigma$ n'est possible que si tous les états de Y ont le même alphabet, et si cet alphabet est indépendant de a. On montrera par la suite que, pour tous les états accessibles de $\mathcal{B}$, tous les états de la composante Y ont bien le même alphabet. Une telle transition prend alors la forme suivante :

$$(P, X, E, Y) \xrightarrow{a} (P', X, E \odot a, Y)$$

où $P' = P \bullet a \cup \{(X \cdot a, E)\} \cup \{(Z, G \odot a) \mid (Z, G) \in P \text{ et } (a, Z) \in I\}$; la notation (abusive) $(a, Z) \in I$ signifie que tous les éléments de Z doivent avoir le même alphabet, et que cet alphabet doit être indépendant de la lettre a.

Cette fois, les ensembles X et Y demeurent inchangés, alors qu'on met à jour l'ensemble E et l'ensemble P. En ce qui concerne ce dernier, on fait d'abord agir a sur tous les (Z, G) comme lors d'une γ-transition (partie $P \bullet a$

de la formule), on ajoute à P une nouvelle vérification qui prend en compte les lettres ayant déjà engendrées des γ-transitions (partie $\{(X \cdot a, E)\}$), et enfin on garde les vérifications (Z, G) de P dont l'alphabet commute avec a (dernière partie de la formule), tout en mettant à jour les composantes Z en $Z \odot a$.

Changement de phases

Lors de la deuxième phase, on suppose qu'on a déjà lu toutes les lettres de x, et donc lancé toutes les vérifications correspondantes pour β. Il ne reste donc plus qu'à s'assurer que ces vérifications sont bien satisfaites, de même que la vérification de γ lancée dans l'ensemble Y. Comme on doit s'assurer du succès de chaque vérification (Z, G) de P, il est nécessaire de passer d'une vue ensembliste à une vue ordonnée, afin d'expliciter la condition de Büchi généralisée qui convient. Pour chaque (Z, G) de P, on est donc amené à choisir un état p de Z qui provienne de l'automate $\mathcal{A}_\beta^G$, c'est-à-dire qui soit dans Q_1^G. Une fois ces états choisis pour tous les (Z, G) de P, on les ordonne. Comme il y en a au plus $2^{|Q_1||\Sigma|}$, on comble les vides avec un symbole externe, $\#$. Posons $K = 2^{|Q_1||\Sigma|}$.

L'ensemble des états de l'automate $\mathcal{C}$ est donc

$$Q_\mathcal{C} = (Q_1 \cup \{\#\})^{\times K} \times Q_2,$$

où $(Q_1 \cup \{\#\})^{\times K}$ est un produit de K copies de $Q_1 \cup \{\#\}$.

De tout état (P, X, E, Y) de l'automate $\mathcal{B}$ on peut donc effectuer une ϵ-transition (transition d'étiquette nulle) vers un état (V, q) de l'automate $\mathcal{C}$, où $q \in Q_2$ et $V = (q_1, q_2, \ldots q_K)$ de la façon suivante :

- Pour tout $(Z, G) \in P$, on choisit un $q_{(Z,G)}$ dans $Z \cap Q_1^G$.
- On ordonne tous ces $q_{(Z,G)}$ suivant un ordre quelconque et on les range dans $(q_1, q_2, \ldots q_j)$ avec $j \leq K$.
- On complète en posant $q_i = \#$ pour tout i tel que $j < i \leq K$.
- Enfin, on choisit un q dans $Y \cap Q_2^E$.

Les transitions de $\mathcal{C}$ sont de la forme :

$$(V, q) \xrightarrow{a} (V', q')$$

où

- Pour tout q_i de V, $q_i \xrightarrow{a}_1 q_i'$ dans $\mathcal{A}_1$ si $q_i \neq \#$, et $q_i' = q_i = \#$ sinon.
- $q \xrightarrow{a}_2 q'$ dans $\mathcal{A}_2$.

Une exécution dans $\mathcal{C}$ doit être acceptante si les exécutions induites sur chacune des composantes le sont. Il faut de plus prendre garde à l'ajout du symbole # qui n'a pas de signification dans $\mathcal{A}_1$. Notons F_i et R_i les conditions d'acceptation de l'automate $\mathcal{A}_i$ pour les mots finis et infinis, pour $i = 1, 2$. Ces conditions (qui correspondent à une union disjointe d'automates pour différents contextes de présent) se construisent naturellement, comme dans le cas de la disjonction de deux formules.

Les conditions d'acceptation de l'automate $\mathcal{C}$ sont alors :

$$F_{\mathcal{C}} = (F_1 \cup \{\#\})^{\times K} \times F_2,$$

et

$$R_{\mathcal{C}} = \bigcup_{r_1 \in R_1} \bigcup_{r_2 \in R_1} \cdots \bigcup_{r_K \in R_1} \bigcup_{r \in R_2} (r_1^{\#} \times r_2^{\#} \times \cdots \times r_K^{\#} \times r),$$

où pour tout i, $r_i^{\#} = r \cup \{\#\}$.

Ces conditions garantissent que l'exécution est acceptante dans $\mathcal{C}$ si chacune des composantes correspond soit à une exécution acceptante, soit à une suite constante de #.

5.7 Preuves

On introduit dans cette section des notations utiles pour prouver que l'automate $\mathcal{A}_\psi^B$ construit à la section précédente reconnaît bien les linéarisations des traces u vérifiant $u, B \models \psi = \beta \,\mathcal{U}\, \gamma$.

Considérons une exécution partielle

$$(P_0, X_0, E_0, Y_0) \xrightarrow{a_1} (P_1, X_1, E_1, Y_1) \xrightarrow{a_2} \ldots \xrightarrow{a_k} (P_k, X_k, E_k, Y_k)$$

dans $\mathcal{B}$ sur un préfixe fini $a_1 a_2 \ldots a_k$ d'un mot w, où $(P_0, X_0, E_0, Y_0) = (\emptyset, S_1, B, S_2)$ est l'état initial de l'automate $\mathcal{B}$. On note r un sous-mot de $a_1 a_2 \ldots a_k$ constitué de lettres sur lesquelles on a effectué des β-transitions, et on note s le sous-mot constitué des autres lettres de $a_1 a_2 \ldots a_k$. Notons que, les β-transitions et les γ-transitions n'étant pas toujours syntaxiquement distinguables, il se peut qu'un tel couple (r, s) ne soit pas unique, pour une exécution partielle fixée. Cette ambiguité potentielle n'est pas problématique. On suppose à partir de maintenant qu'on a fixé un tel couple (r, s).

Lemme 8 *Soit $t = \varphi(a_1 a_2 \dots a_k)$ la trace associée au mot $a_1 a_2 \dots a_k$. Alors les assertions suivantes sont vérifiés :*

1. $t = \varphi(r)\varphi(s)$.
2. $\mathrm{alph}(Y_k) = \mathrm{alph}(X_k) = \mathrm{alph}(s)$.
3. $E_k = B \odot r$.
4. $Y_k = S_2 \cdot s$.
5. $X_k = S_1 \cdot s$.

Preuve. On raisonne par induction sur la longueur k de l'exécution partielle considérée. Si $k = 0$, alors $t = \epsilon$ et $r = s = \epsilon$, donc tous les points sont triviaux sachant que $(P_0, X_0, E_0, Y_0) = (\emptyset, S_1, B, S_2)$.

Supposons le Lemme vérifié pour $k-1$, et montrons qu'il l'est encore pour k. On note respectivement t_i, r_i et s_i la trace et les deux sous-mots associés à l'exécution partielle sur $a_1 a_2 \dots a_i$.

Cas 1: a_k correspond à une γ-transition Par conséquent $r_k = r_{k-1}$ et $s_k = s_{k-1} a_k$.

1. $t_k \stackrel{\text{def}}{=} t_{k-1}\varphi(a_k) \stackrel{ind}{=} \varphi(r_{k-1})\varphi(s_{k-1})\varphi(a_k) = \varphi(r_{k-1})\varphi(s_{k-1}a_k)$ $= \varphi(r_k)\varphi(s_k)$, donc $t = \varphi(r)\varphi(s)$.
2. $\mathrm{alph}(Y_k) \stackrel{\text{def}}{=} \mathrm{alph}(Y_{k-1}) \cup \{a_k\} \stackrel{ind}{=} \mathrm{alph}(s_{k-1}) \cup \{a_k\} = \mathrm{alph}(s_k) = \mathrm{alph}(s)$. De même, $\mathrm{alph}(X_k) = \mathrm{alph}(s)$.
3. $E_k \stackrel{\text{def}}{=} E_{k-1} \stackrel{ind}{=} B \odot r_{k-1} = B \odot r_k = B \odot r$.
4. $Y_k \stackrel{\text{def}}{=} Y_{k-1} \cdot a_k \stackrel{ind}{=} (S_2 \cdot s_{k-1}) \cdot a_k = S_2 \cdot s_k = S_2 \cdot s$.
5. $X_k \stackrel{\text{def}}{=} X_{k-1} \cdot a_k \stackrel{ind}{=} (S_1 \cdot s_{k-1}) \cdot a_k = S_1 \cdot s_k = S_1 \cdot s$.

Cas 2: a_k correspond à une β-transition Par conséquent $r_k = r_{k-1}a_k$ et $s_k = s_{k-1}$.

1. a_k correspondant à une β-transition, on a nécessairement $(a_k, Y_{k-1}) \in I$, donc (induction) $(a_k, s_{k-1}) \in I$, c'est-à-dire $(a_k, s_k) \in I$. Ainsi $t = t_k \overset{\text{def}}{=} t_{k-1}\varphi(a_k) \overset{ind}{=} \varphi(r_{k-1})\varphi(s_{k-1})\varphi(a_k) = \varphi(r_{k-1})\varphi(a_k)\varphi(s_{k-1}) = \varphi(r_k)\varphi(s_k) = \varphi(r)\varphi(s)$.

2. $\mathrm{alph}(Y_k) \overset{\text{def}}{=} \mathrm{alph}(Y_{k-1}) \overset{ind}{=} \mathrm{alph}(s_{k-1}) = \mathrm{alph}(s_k) = \mathrm{alph}(s)$. De même, $\mathrm{alph}(X_k) = \mathrm{alph}(s)$.

3. $E_k \overset{\text{def}}{=} E_{k-1} \odot a_k \overset{ind}{=} (B \odot r_{k-1}) \odot a_k = B \odot (r_{k-1}a_k) = B \odot r_k = B \odot r$.

4. $Y_k \overset{\text{def}}{=} Y_{k-1} \overset{ind}{=} S_2 \cdot s_{k-1} = S_2 \cdot s_k = S_2 \cdot s$.

5. $X_k \overset{\text{def}}{=} X_{k-1} \overset{ind}{=} S_1 \cdot s_{k-1} = S_1 \cdot s_k = S_1 \cdot s$.

□

Ce lemme montre que si (P, X, E, Y) est un état accessible dans $\mathcal{B}$ après la lecture d'un mot de $r \sqcup_I s$, alors tous les états de X et de Y ont le même alphabet, qui est aussi celui de r. De plus, E est bien le contexte de présent $B \odot r$, et enfin X et Y sont les états accessibles dans $\mathcal{A}_1$ et $\mathcal{A}_2$ (respectivement) après lecture du mot s.

On va maintenant associer à chaque (Z, G) de P_k un ensemble de mots $f_k(Z, G)$ qui correspond à une classe de I-suffixes équivalents de r. Pour ce faire, on va définir un ensemble de fonctions f_i de manière inductive.

On définit des fonctions $f_i : P_i \to 2^{\Sigma^*}$, pour $i \in \{1, 2, \ldots, k\}$ comme suit : pour tout i, $0 < i < k$,

- Si a_i correspond à une γ-transition dans cette exécution, pour tout $(Z', G') \in P_{i+1}$,

$$f_{i+1}(Z', G') = \bigcup_{\substack{(Z,G') \in P_i \\ Z' = Z \cdot a_i}} f_i(Z, G').$$

  Dans ce cas en effet, $a_i \in s$ donc il n'y a pas de nouvel I-suffixe de r, mais les (Z, G') de P_i ont été transformés en $(Z \cdot a_i, G')$ par la γ-transition.

- Si a_i correspond à une β-transition dans cette exécution, pour tout $(Z', G') \in P_{i+1}$,

$$f_{i+1}(Z', G') = \bigcup_{\substack{(Z,G')\in P \\ Z'=Z\cdot a_i}} (f_i(Z, G').a_i) \cup \bigcup_{\substack{(Z',G)\in P \\ G'=G\odot a_i \\ (a_i,Z)\in I}} f_i(Z', G) \cup \bigcup_{\text{si } (Z',G')=(X_i\cdot a_i,E_i)} \{a_i\}$$

  Dans ce cas, on a $a_i \in r$, et les nouveaux I-suffixes de r s'obtiennent par application du Lemme de Levi : on ajoute a_i en fin des I-suffixes existant, on garde les I-suffixes existant qui commutent avec a_i, et enfin on prend $\{a_i\}$ comme nouvel I-suffixe.

La correspondance entre les (Z, G) de P_k et les I-suffixes de r est assurée par le lemme suivant :

Lemme 9 *1. Pour tout I-suffixe non-vide r'' de r, il existe un $(Z, G) \in P_k$ tel que $r'' \in f_k(Z, G)$.*

2. Pour tout $(Z, G) \in P_k$, et pour tout $r'' \in f_k(Z, G)$, r'' est un I-suffixe non-vide de r; de plus, si r' est un mot fini tel que $\varphi(r')\varphi(r'') = \varphi(r)$ alors $G = B \odot r'$ et $Z = S_1 \cdot (r''s)$.

Preuve. On raisonne par induction sur la longueur k de l'exécution partielle. Si $k = 0$, alors $r = \epsilon$ et $P_0 = \emptyset$, donc les deux points sont trivialement vérifiés.

Supposons le Lemme vérifié pour $k - 1$, et montrons qu'il l'est encore pour k.

Cas 1: a_k correspond à une γ-transition. Par conséquent $r_k = r_{k-1}$ et $s_k = s_{k-1}a_k$.

1. Soit $r'' \neq \epsilon$ tel que $\varphi(r')\varphi(r'') = \varphi(r) = \varphi(r_k) = \varphi(r_{k-1})$. Alors (induction) il existe $(Z, G) \in P_{k-1}$ tel que $r'' \in f_{k-1}(Z, G)$. Or $f_{k-1}(Z, G) \subseteq f_k(Z \cdot a_k, G)$ et $(Z \cdot a_k, G) \in P_k$.

2. Soit $(Z',G') \in P_k$ et soit $r'' \in f_k(Z',G')$. Par définition de $f_k(Z',G')$, il existe $(Z,G') \in P_{k-1}$ tel que $Z = Z' \cdot a_k$ vérifiant $r'' \in f_{k-1}(Z,G')$. Alors (induction) il existe $r' \in \Sigma^*$ tel que $\varphi(r_{k-1}) = \varphi(r')\varphi(r'')$, avec $r'' \neq \epsilon$, $G' = B \odot r'$ et $Z = S_1 \cdot (r'' s_{k-1})$. On a donc également $\varphi(r_k) = \varphi(r')\varphi(r'')$, $G' = B \odot r'$ et $Z' = (S_1 \cdot r'' s_{k-1}) \cdot a_k = S_1 \cdot (r'' s_k) = S_1 \cdot (r'' s)$.

Cas 2: a_k correspond à une β-transition. Par conséquent $r_k = r_{k-1} a_k$ et $s_k = s_{k-1}$.

1. Soit $r'' \neq \epsilon$ tel que $\varphi(r')\varphi(r'') = \varphi(r) = \varphi(r_{k-1}a_k) = \varphi(r_{k-1})\varphi(a_k)$. Par application du Lemme de Levi, deux cas se présentent : soit il existe $x \in \Sigma^*$ tel que $\varphi(r') = \varphi(x)\varphi(a_k)$, $\varphi(r_{k-1}) = \varphi(x)\varphi(r'')$ et $(a_k, r'') \in I$, soit il existe $x \in \Sigma^*$ tel que $\varphi(r'') = \varphi(x)\varphi(a_k)$, et $\varphi(r_{k-1}) = \varphi(r')\varphi(x)$.

   Cas 2.1: $\varphi(r') = \varphi(x)\varphi(a_k)$, $\varphi(r_{k-1}) = \varphi(x)\varphi(r'')$ et $(a_k, r'') \in I$. Comme $\varphi(r_{k-1}) = \varphi(x)\varphi(r'')$, avec $r'' \neq \epsilon$, il existe (induction) un $(Z,G) \in P_{k-1}$ tel que $r'' \in f_k(Z,G)$. Alors (induction) $Z = S_1 \cdot (r'' s_{k-1})$ et $G = B \odot x$. Comme $(s_{k-1}, a_k) \in I$ et $(r'', a_k) \in I$ on a aussi $(r'' s_{k-1}, a_k) \in I$, donc $(Z, a_k) \in I$, ce qui implique $(Z, G \odot a_k) \in P_k$. Or $r'' \in f_{k-1}(Z,G) \subseteq f_k(Z, G \odot a_k)$.

   Cas 2.2: $\varphi(r'') = \varphi(x)\varphi(a_k)$ et $\varphi(r_{k-1}) = \varphi(r')\varphi(x)$. Si $x \neq \epsilon$, alors (induction) il existe $(Z,G) \in P_{k-1}$ tel que $x \in f_{k-1}(Z,G)$. Or $f_{k-1}(Z,G)a_k \subseteq f_k(Z \cdot a_k, G)$ et $(Z \cdot a_k, G) \in P_k$. Donc $xa_k \in f_k(Z \cdot a_k, G)$. Comme r'' est un sous-mot de $r = r_{k-1}a_k$, et comme $\varphi(r'') = \varphi(x)\varphi(a_k)$, on en déduit que r'' se termine par a_k. On a donc nécessairement $r'' = ya_k$, avec $\varphi(x) = \varphi(y)$. On peut donc substituer y à x dans les déductions précédentes, ce qui implique en particulier que $ya_k \in f_k(Z \cdot a_k, G)$, c'est-à-dire $r'' \in f_k(Z \cdot a_k, G)$.

   Si $x = \epsilon$, alors $r'' = a_k$ et $r_{k-1} = r'$. Or $(X_{k-1} \cdot a_k, E_{k-1}) \in P_k$ et $\{a_k\} \subseteq f_k(X_{k-1} \cdot a_k, E_{k-1})$, donc $r'' \in f_k(X_{k-1} \cdot a_k, E_{k-1})$.

2. Soit $(Z',G') \in P_k$ et $r'' \in f_k(Z',G')$. Comme

$$f_k(Z',G') = \bigcup_{\substack{(Z,G') \in P \\ Z' = Z \cdot a_k}} (f_{k-1}(Z,G').a_k) \cup \bigcup_{\substack{(Z',G) \in P \\ G' = G \odot a_k \\ (a_k, Z') \in I}} f_{k-1}(Z',G)$$

$$\cup \bigcup_{\text{si } (Z',G')=(X_{k-1}a_k,E_{k-1})} \{a_k\}$$

trois cas se présentent.

Cas 2.a: $r'' \in f_{k-1}(Z,G').a_k$ **pour un** $(Z,G') \in P_{k-1}$ **tel que** $Z' = Z \cdot a_k$**.** Donc $r'' = xa_k$, avec $x \in f_{k-1}(Z,G')$. Alors (induction) il existe r' tel que $\varphi(r_{k-1}) = \varphi(r')\varphi(x)$, $x \neq \epsilon$, $G' = B \odot r'$ et $Z = S_1 \cdot (xs_{k-1})$. Ainsi, $vp(r) = \varphi(r_{k-1})\varphi(a_k) = \varphi(r')\varphi(x)\varphi(a_k) = \varphi(r')\varphi(r'')$, $r'' \neq \epsilon$, $G' = B \odot r'$ et $Z' = (S_1 \cdot (xs_{k-1})) \cdot a_k = S_1 \cdot (xs_{k-1}a_k)$. De plus, a_k correspondant à une β-transition, on a $(a_k, s_{k-1}) \in I$, donc $Z' = S_1 \cdot (xa_ks_{k-1}) = S_1 \cdot (r''s)$.

Cas 2.b: $r'' \in f_{k-1}(Z',G)$ **pour un** $(Z',G) \in P_{k-1}$ **tel que** $G' = G \odot a_k$ **et** $(a_k, Z') \in I$**.** Alors il existe (induction) un r' tel que $\varphi(r_{k-1}) = \varphi(r')\varphi(r'')$, $r'' \neq \epsilon$, $G = B \odot r'$ et $Z' = S_1 \cdot (r''s_{k-1})$. Alors $\varphi(r) = \varphi(r_{k-1})\varphi(a_k) = \varphi(r')\varphi(r'')\varphi(a_k) = \varphi(r')\varphi(a_k)\varphi(r'') = \varphi(r'a_k)\varphi(r'')$ car $(a_k, Z') \in I$ et $\text{alph}(Z') = \text{alph}(r''s_{k-1})$. De plus, $G' = G \odot a_k = (B \odot r') \odot a_k = B \odot (r'a_k)$, et enfin $Z' = S_1 \cdot (r''s)$.

Cas 2.c: $r'' = \{a_k\}$ **et** $(Z',G') = (X_{k-1}a_k, E_{k-1})$**.** Alors $r = r_{k-1}a_k = r_{k-1}r''$, donc $\varphi(r) = \varphi(r_{k-1})\varphi(r'')$, $r'' \neq \epsilon$, $G' = E_{k-1} = B \odot (r_{k-1}s_{k-1}) = B \odot (r_{k-1}s_k)$, et $Z' = X_{k-1}a_k = (S_1 \cdot s_{k-1}) \cdot a_k = S_1 \cdot (s_{k-1}a_k)$. Comme a_k et s_{k-1} commutent, $Z' = S_1 \cdot (a_ks_{k-1}) = S_1 \cdot (r''s)$.

□

Ce dernier lemme va nous permettre de prouver plus aisément les deux théorèmes principaux de cette section, qui montrent que l'automate construit pour l'opérateur *until* a bien les propriétés attendues :

Théorème 19 *Soit $w \in \Sigma^\infty$ et $t = \varphi(w)$. Si t appartient à $L_B(\beta\,\mathcal{U}\,\gamma)$, alors w est reconnu par $\mathcal{A}^B_{\beta\mathcal{U}\gamma}$.*

Preuve. Puisque t appartient à $L_B(\beta\,\mathcal{U}\,\gamma)$, on a $B, t \models \beta\,\mathcal{U}\,\gamma$. Il existe donc $u \in \mathbf{M}(\Sigma, D)$ et $v \in \mathbf{R}(\Sigma, D)$ tels que $t = uv$, $B \odot u, v \models \gamma$, et pour tout $u'u'' = u$, si $u'' \neq \epsilon$, alors $B \odot u', u''v \models \beta$.

Comme $\varphi(w) = t = uv$, il existe un sous-mot fini x de w, et un sous-mot y de w tels que $\varphi(x) = u$ et $\varphi(y) = v$. Le mot w appartient au I-shuffle de x et y.

Par conséquent, il existe un préfixe fini w_1 de w contenant entièrement le sous-mot x. De plus, $\varphi(x) = u$ est un préfixe (de traces) de $\varphi(w_1)$. On a donc $\varphi(w_1) = uv_1$ pour un certain préfixe v_1 de v. On pose $w = w_1w_2$, et on note z le sous-mot de w_1 tel que $w_1 \in x \sqcup\!\sqcup_I z$. De plus, on a $y = zw_2$. Comme $w_1 \in x \sqcup\!\sqcup_I z$, toute lettre de w_1 appartenant à x est indépendante de toutes les lettres antérieures (i.e. de positions plus petites) de w_1 qui appartiennent à y.

On choisit alors de faire exécuter tout le préfixe w_1 dans l'automate $\mathcal{B}$, en effectuant des β-transitions sur les lettres de x et des γ-transitions sur les lettres de z. Une telle exécution est possible compte tenu de la remarque précédente sur les relations d'indépendance entre les lettres de x et celles de z. Posons $s = \varphi(z)$. La trace t s'écrit alors $t = uv = us\varphi(w_2)$.

Une telle exécution partielle conduit à un état (P, X, E, Y). Par application des lemmes précédents, cet état satisfait les propriétés suivantes:

- $X = S_1 \cdot z$;
- $Y = S_2 \cdot z$;
- $E = B \odot x$;
- pour tout $(Z, G) \in P$, il existe $x', x'' \in \Sigma^*$, tels que $\varphi(x) = \varphi(x')\varphi(x'')$, x'' est non-vide, $G = B \odot x'$ et $Z = S_1 \cdot (x''z)$.

On choisit alors d'effectuer une ϵ-transition de cet état (P, X, E, Y) vers un état (V, q) de l'automate $\mathcal{C}$. Comme par hypothèse on a $B \odot u, v \models \gamma$, on a également $B \odot x, s\varphi(w_2) \models \gamma$. Le mot zw_2 étant une linéarisation de la trace $s\varphi(w_2)$, ce mot est accepté par l'automate $\mathcal{A}_\gamma^{B \odot x}$. Il existe donc un état $q \in S_2 \cdot z$ de cet automate tel qu'il existe une exécution acceptante du mot w_2 à partir de cet état. Ceci montre que l'on peut choisir $q \in S_2 \cdot z \cap Q_2^{B \odot x} = Y \cap Q_2^E$ lors de l'ϵ-transition vers $\mathcal{C}$, et que de cet état on peut mener une exécution acceptante dans l'automate $\mathcal{A}_\gamma$.

De même, considérons un couple (Z, G) de P. Il existe alors $x', x'' \in \Sigma^*$, avec $x'' \neq \epsilon$ tels que $u = \varphi(x) = \varphi(x')\varphi(x'')$, $Z = S_1 \cdot (x''z)$ et $G = B \odot x'$. Comme par hypothèse on a $B \odot \varphi(x'), \varphi(x'')s\varphi(w_2) \models \beta$, on a également $B \odot x', \varphi(x''zw_2) \models \beta$. Le mot $x''zw_2$ est donc accepté par l'automate $\mathcal{A}_\beta^{B \odot x'}$.

Il existe donc un état $p \in S_1 \cdot (x''z)$ de cet automate tel qu'il existe une exécution acceptante du mot w_2 à partir de cet état. On peut donc choisir un $p \in S_1 \cdot (x''z) \cap Q_1^{B \odot x'} = Z \cap Q_1^G$ lors de l'ϵ-transition vers $\mathcal{C}$ tel que de cet état on puisse mener une exécution acceptante dans l'automate $\mathcal{A}_\beta$.

Le mot $w = w_1 w_2$ est par conséquent accepté par l'automate $\mathcal{A}^B_{\beta\mathcal{U}\gamma}$. □

Le théorème suivant établit la réciproque :

Théorème 20 *Soit $w \in \Sigma^\infty$ et $t = \varphi(w)$. Si w est reconnu par $\mathcal{A}^B_{\beta\mathcal{U}\gamma}$, alors t appartient à $L_B(\beta\,\mathcal{U}\,\gamma)$.*

Preuve. Soit ρ une exécution acceptante de $\mathcal{A}^B_{\beta\mathcal{U}\gamma}$ sur le mot w. Soient $w_1 \in \Sigma^*$ et $w_2 \in \Sigma^\infty$ tels que $w = w_1 w_2$ et tels que l'unique ϵ-transition de ρ intervienne entre les mots w_1 et w_2. L'exécution ρ s'effectuant dans $\mathcal{B}$ sur w_1, soit r le sous-mot de w_1 correspondant à des β-transitions dans $\mathcal{B}$, et soit s le sous-mot de w_1 complémentaire, i.e. correspondant à des γ-transitions dans $\mathcal{B}$. D'après la garde des β-transitions, on sait que toute lettre de w_1 appartenant à r est indépendante de toutes les lettres de w_1 antérieures (i.e. de positions plus petites) appartenant à s. Le mot w_1 est un I-shuffle de r et de s.

Soit (P, X, E, Y) l'état de l'automate $\mathcal{B}$ atteint après l'exécution de $\mathcal{B}$ sur le mot $w_1 \in r \sqcup_I s$. Par application des lemmes précédents, cet état vérifie les propriétés suivantes:

- $X = S_1 \cdot s$;
- $Y = S_2 \cdot s$;
- $E = B \odot r$;
- Pour tout sous-mot non-vide r'' de r tel qu'il existe un $r' \in \Sigma^*$ vérifiant $\varphi(r')\varphi(r'') = \varphi(r)$, il existe un $(Z, G) \in P$ tel que $G = B \odot r'$ et $Z = S_1 \cdot (r''s)$.

Posons $u = \varphi(r)$, et $v = \varphi(sw_2)$. Montrons d'abord que $B \odot u, v \models \gamma$, i.e. que $B \odot r, \varphi(s)\varphi(w_2) \models \gamma$. Il suffit pour cela de montrer que la linéarisation sw_2 est acceptée par l'automate $\mathcal{A}_\gamma^{B \odot r}$. Ceci équivaut à montrer qu'il existe un état $q \in S_2 \cdot s \cap Q_2^{B \odot r}$ à partir duquel on peut mener une exécution acceptante dans $\mathcal{A}_\gamma$ sur le mot w_2. Or, l'exécution ρ étant acceptante, un tel état existe puisque $S_2 \cdot s \cap Q_2^{B \odot r} = Y \cap Q_2^E$.

Soit maintenant un suffixe de trace u'' non-vide de u. Posons $u = u'u''$. Il existe alors des sous-mots r' et r'' de r tels que $\varphi(r') = u'$ et $\varphi(r'') = u''$. De plus, on a naturellement $\varphi(r')\varphi(r'') = u'u'' = u = \varphi(r)$. Comme u'' n'est pas vide, r'' ne l'est pas non plus. On en conclut qu'il existe un (Z, G) dans P tel que $G = B \odot r'$ et $Z = S_1 \cdot (r''s)$. Pour montrer que $B \odot u', u''v \models \beta$, il suffit de montrer que $B \odot r', \varphi(r''sw_2) \models \beta$, c'est-à-dire que le mot $r''sw_2$ est accepté par l'automate $\mathcal{A}_\beta^{B\odot r'}$. Ceci équivaut à montrer qu'il existe un état $p \in S_1 \cdot (r''s) \cap Q_1^{B\odot r}$ à partir duquel on peut mener une exécution acceptante dans $\mathcal{A}_\beta$ sur le mot w_2. Or, l'exécution ρ étant acceptante, un tel état existe puisque $S_1 \cdot (r''s) \cap Q_1^{B\odot r'} = Z \cap Q_1^G$. $\square$

5.8 Conclusion

Nous avons montré dans ce chapitre que l'on peut effectivement construire, pour toute formule ψ de LTrL et tout ensemble compatible B, un automate de Büchi généralisé qui reconnaît exactement les linéarisations des traces du langage $L_B(\psi)$. En prenant $B = \emptyset$, on obtient donc un automate qui reconnaît exactement les linéarisations des traces de $L(\psi)$. La traduction que nous proposons, des formules de LTrL vers les automates de Büchi, est *modulaire*, donc elle permet une certaine réutilisabilité. Comme la logique est non-élémentaire, cette traduction l'est également. Plus précisemment, seules deux structures de formules provoquent une explosion combinatoire : la négation et l'opérateur *until*. Dans ce dernier cas, l'explosion combinatoire a lieu par rapport au membre *gauche* de l'opérateur $\mathcal{U}$. Lorsque l'on cherche à traduire une formule de LTrL en un automate de Büchi en utilisant notre construction, il faut donc tenir compte principalement de la profondeur d'imbrication à gauche des *until* et des négations. En outre, il est rare d'utiliser dans la pratique des formules de spécification présentant une profondeur d'imbrication à gauche des *until* élevée (i.e. supérieure à 2). Il faut donc surveiller plus particulièrement la profondeur d'imbrication des négations. Sous ces deux conditions, on peut espérer obtenir des automates de tailles raisonnables, et par conséquent mener à bien des tâches de vérification telles que le model-checking ou la satisfaisabilité sur des exemples concrets.

Chapitre 6

Graphes de Signaux de Transitions

6.1 Introduction

On s'intéresse dans ce chapitre au problème du model checking pour une classe particulière de graphes. Une application naturelle de la logique temporelle pour les traces LTrL est la spécification des systèmes distribués possédant une sémantique concurrente, comme par exemple les réseaux de Petri [51]. Malheureusement, on a vu que le problème du model checking pour LTrL peut s'avérer non-élémentaire (cf. Chapitre 5). Afin de conserver une certaine faisabilité (c'est-à-dire une complexité raisonnable), on est donc amené à restreindre la classe des réseaux que l'on considère. Une classe suggérée naturellement par la littérature sur les réseaux de Petri est celle des réseaux 1-bornés ou sûrs (réseaux dans lesquels les places ne contiennent jamais plus d'un jeton) [5]. Cette classe est notamment utilisée pour modéliser et spécifier des circuits asynchrones, un type particulier d'architecture matérielle, et plus spécifiquement des réseaux de signaux de transitions (signaux booléens échangés entre des composants électroniques) [41, 51]. Pour ces réseaux de signaux de transitions, on peut écrire des spécifications dans la logique LTrL, et nous montrons dans ce chapitre qu'il existe pour le problème du model checking un algorithme dont la complexité en temps est en $O(n^2 \times k^2 \times m)$, où n est la taille de l'espace des états du réseau considéré, k le nombre de transitions dans le graphe sous-jacent, et m est la taille de la formule de spécification.

Ce chapitre établit donc un lien entre les spécifications en logiques temporelles sur les traces et les descriptions des circuits asynchrones. Le choix de la logique LTrL se justifie principalement par sa caractérisation canonique en termes de pouvoir d'expression : cette logique est équivalent à la logique du premier ordre (cf. Chapitres 3 et 5). Quant au choix du modèle, il se justifie par le désir de pouvoir travailler sur un préfixe fini du graphe de dépliage tout en conservant toutes les informations contenues dans le réseau [29, 23, 25, 26]. Les réseaux de signaux de transitions dont le réseau de Petri sous-jacent est un graphe marqué constituent à cet égard un point de départ idéal. Un inconvénient majeur de ce modèle est qu'il ne permet de spécifier que des systèmes fermés, et non des systèmes avec entrées et sorties. Ces résultats ont fait l'objet d'une publication lors du *Workshop on Hardware Design and Petri Nets* de la conférence ICATPN à Williamsburg (Virginie, USA) en juin 1999 [32].

6.2 Graphes de Signaux de Transitions

6.2.1 Architectures asynchrones

L'*architecture matérielle* d'un système informatique est celle de l'ensemble de ses *composants* électroniques, par opposition à l'architecture logicielle, qui concerne les programmes pouvant être exécutés sur ces composants. Un formalisme particulier, les *réseaux de Petri*, constitue un outil particulièrement bien adapté à la modélisation et à la spécification des architectures matérielles dites *asynchrones*, c'est-à-dire dans lesquelles n'interviennent pas d'horloges communes : dès qu'un composant est prêt à changer d'état, il le fait sans attendre le “top” d'une horloge. Les réseaux de Petri présentent par nature un tel asynchronisme. De plus, les réseaux de Petri permettent d'exprimer une réelle *concurrence* entre des événements, ce qui est également crucial pour la modélisation des architectures matérielles : ces architectures pouvant comporter un très grand nombre de composants, il est inenvisageable de prendre en compte des événements concurrents sur un mode d'entrelacement, i.e. via un modèle séquentiel. Ces deux caractéristiques, *concurrence* et *asynchronisme*, font donc des réseaux de Petri un outil de premier choix pour l'analyse de ce type d'architectures.

6.2.2 Circuits asynchrones

Un *circuit asynchrone* peut être vu comme une implémentation *matérielle* d'un progamme parallèle distribué. Cependant, contrairement aux programmes parallèles, qui nécessitent en général l'arbitrage d'un mécanisme de contrôle du temps, les circuits asynchrones ne sont régulés que par le comportement de leurs propres composants : portes logiques, mémoires, etc... Chaque composant possède des fiches d'entrée et de sortie, et les fiches sont reliées entre elles par un réseau de conducteurs.

Au niveau physique, on considère de tels circuits comme *discrets*, c'est-à-dire que les signaux échangés entre les composants sont codés par des variables booléennes, et les changements internes des composants par des transitions de 0 vers 1 et vice versa, appelées respectivement transitions *up* et *down*.

Au niveau logique, le comportement d'un circuit asynchrone est caractérisé par les suites de transitions *up* et *down* aux entrées et aux sorties de ses composants. L'ordre relatif de ces transitions n'est imposé par aucun mécanisme de contrôle, et n'est déterminé que par des relations causales d'ordre local. En raison de cette localité, l'ordre de l'ensemble des événements est *partiel*, et non total.

6.2.3 Réseaux de Petri et $\mathcal{T}$-systèmes

Un *réseau de Petri* est un triplet (P, T, F), où P est un ensemble fini de *places*, T un ensemble fini de *transitions* (distinct de P), et $F \subseteq (P \times T) \cup (T \times P)$ une relation de *flux* qui lie les places et les transitions. On peut représenter graphiquement un réseau de Petri de la façon suivante : les places sont représentées par des cercles, les transitions par des carrés, et la relation de flux par des flèches.

Exemple 12 *Sur la Figure 6.1, on a représenté un réseau de Petri à 8 places et 4 transitions.*

L'ensemble $P \cup T$ est appelé l'ensemble des *éléments* du réseau. Pour tout élément x du réseau, on définit les ensembles pre-set ${}^\bullet x$ et post-set $x^\bullet$ de x comme suit :

$${}^\bullet x = \{y \in (P \cup T) \mid (y, x) \in F\},$$

et

$$x^\bullet = \{y \in (P \cup T) \mid (x, y) \in F\}.$$

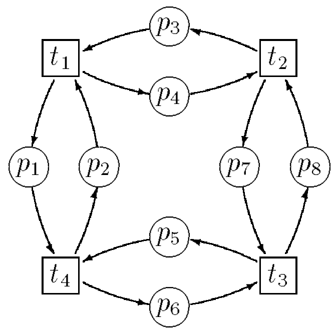

FIG. 6.1 – *Un réseau de Petri*

Exemple 13 *Sur l'exemple de la Figure 6.1, on a :*

$$^\bullet p_1 = \{t_1\} \textit{ et } t_1^\bullet = \{p_1, p_4\}.$$

Un *marquage* d'un réseau (P, T, F) est une application $M : P \to \mathbf{N}$ qui associe à chaque place un entier naturel. Pour une place $p \in P$, on dit que p contient k *jetons* si $M(p) = k$. Une place p est *marquée* par un marquage M si $M(p) > 0$ (i.e. si elle n'est pas vide).

Une transition $t \in T$ est *franchissable* dans un marquage M si toutes les places de $^\bullet t$ sont marquées par M. Dans ce cas, t peut être *franchie* et ce franchissement transforme le marquage M en un marquage M' défini par :

$$\forall p \in P, M'(p) = \begin{cases} M(p) - 1 & \text{si } p \in {}^\bullet t \setminus t^\bullet, \\ M(p) + 1 & \text{si } p \in t^\bullet \setminus {}^\bullet t, \\ M(p) & \text{sinon.} \end{cases}$$

Remarquons que l'on peut également définir le nouveau marquage M' par l'équation équivalente suivante :

$$\forall p \in P, M'(p) = M(p) - |F \cap \{(p, t)\}| + |F \cap \{(t, p)\}|.$$

Un réseau (P, T, F) muni d'un marquage *initial* M_{in} est appelé un *réseau marqué*, et définit ainsi un système de transitions dont les états sont les marquages du réseau, M_{in} étant l'état initial. On note RM_N (pour *Reachable Markings*) l'ensemble des marquages atteignables à partir de M_{in}. Si M et M' sont dans RM_N, et s'il existe une transition t franchissable en M telle que son franchissement conduise en M', alors on note $M \xrightarrow{t} M'$.

Un réseau marqué (P, T, F, M_{in}) est k-borné si pour tout marquage atteignable à partir de M_{in}, aucune place ne contient plus de k jetons. Un réseau (P, T, F, M_{in}) est *sûr* s'il est 1-borné. Dans ce cas, on peut toujours identifier un marquage atteignable avec le sous-ensemble des places non-vides pour ce marquage. On considère alors les marquages M comme des applications de P dans $\{0, 1\}$.

Exemple 14 *Sur la Figure 6.2, on a représenté un réseau marqué* 1*-borné avec son marquage initial (les jetons sont représentés par des ronds dans les places). Les transitions franchissables dans ce marquage sont t_2 et t_4.*

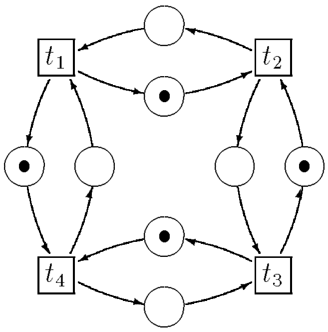

FIG. 6.2 – *Un réseau* 1*-borné et son marquage initial*

Un réseau marqué $N = (P, T, F, M_{in})$ est :

- *fortement connexe* si tout couple d'éléments (x, y) de $P \cup T$ appartient à la fermeture transitive de la relation de flux F, i.e. si le graphe de F est fortement connexe.

- *vivace* si pour toute transition t et tout marquage atteignable M_1, on peut atteindre à partir de M_1 un marquage M_2 dans lequel la transition t est franchissable.

Un $\mathcal{T}$-*système* est un réseau marqué $N = (P, T, F, M_{in})$ vérifiant :

$$\forall p \in P, |{}^\bullet p| = |p^\bullet| = 1.$$

Dans un tel réseau, chaque place est reliée à exactement une transition en entrée et une transition en sortie. Cela n'empêche pas que certaines transitions aient plusieurs places en entrée et/ou en sortie.

6.2.4 Graphes de Signaux de Transitions (GST)

Un *graphe de signaux de transitions* est une structure

$$G = (N, Y, \phi, \mathcal{V}_0)$$

où $N = (P, T, F, M_{in})$ est un $\mathcal{T}$-système *fortement connexe*, *vivace* et *sûr*, Y est un ensemble fini dont les éléments sont appelés des signaux booléens, $\phi : T \to Y \times \{+, -\}$ est une application qui assigne un *signal de transition* à chaque transition de T, et $\mathcal{V}_0 : Y \to \{0, 1\}$ est la *valuation initiale*.

Dans la suite de ce chapitre, on utilisera la notation suivante : $\star$ désigne un élément de $\{+, -\}$, et un signal de transition $(y, \star)$ sera noté $y^\star$.

Exemple 15 *La Figure 6.3 représente un GST simple pour lequel* $Y = \{y_1, y_2\}$ *et dont la valuation initiale est* $\mathcal{V}(y_1) = 0$ *et* $\mathcal{V}(y_2) = 1$.

Remarque 8 *Le caractère* 1*-borné (ou sûr) d'un réseau* N *sous-jacent à un GST* $G = (N, Y, \phi, \mathcal{V}_0)$ *implique que, si une transition* t *est franchissable dans un marquage atteignable* M*, alors toutes les places de* $t^\bullet \setminus {}^\bullet t$ *sont vides pour* M *(sinon une place en sortie pourrait contenir plus d'un jeton après le franchissement de* t*). Le franchissement de* t *consiste alors à "vider" les places de* ${}^\bullet t$ *et "remplir" (avec un jeton) les places de* $t^\bullet$.

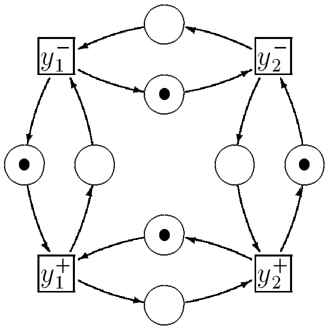

FIG. 6.3 – *Un exemple simple de GST*

Remarque 9 *Si deux transitions distinctes t et t' sont franchissables dans un même marquage atteignable M d'un GST, alors elles sont* indépendantes*, au sens où :*

$$({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset.$$

En effet, une place p de ${}^\bullet t$ vérifie $|p^\bullet| = 1$, donc $p^\bullet = \{t\}$, ce qui implique que $p \notin {}^\bullet t'$. Comme p est pleine pour M (puisque t est franchissable), elle ne peut pas appartenir à $t'^\bullet$ qui ne contient que des places vides. De même, on montre que toute place p de $t^\bullet$ n'est pas dans ${}^\bullet t' \cup t'^\bullet$.

6.3 GST et Traces

Les remarques précédentes montrent que l'on peut, pour un GST donné $G = (N, Y, \phi, \mathcal{V}_0)$ avec $N = (P, T, F, M_{in})$, définir une relation de *dépendance* $D_N \subseteq T \times T$:

$$\forall t, t' \in T, \quad t\ D_N\ t' \text{ ssi } ({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) \neq \emptyset.$$

Alors (T, D_N) constitue un *alphabet de dépendance* au sens des traces de Mazurkiewicz, i.e. la relation D_N est réflexive et symmétrique. On note I_N

la relation d'*indépendance* associée, i.e.

$$I_N = T \times T \setminus D_N.$$

On a vu précedemment que, pour tout marquage atteignable M, les transitions franchissables en M sont deux à deux indépendantes. Pour un tel marquage, on note $t(M)$ l'ensemble des transitions franchissables en M et $u(M)$ la *trace* finie constituée de ces transitions. On va pouvoir associer au GST G une trace TR_G sur l'alphabet de dépendance (T, D_N) de la façon suivante : on pose $M_0 = M_{in}$, et pour tout entier $k \in \mathbf{N}$ on définit M_{k+1} comme le marquage atteint à partir de M_k en franchissant une fois toutes les transitions de $t(M_k)$.

On définit alors la trace infinie

$$\mathrm{TR}_G = u(M_0)u(M_1)u(M_2)\ldots u(M_k)\ldots$$

par concaténation, et on appelle TR_G la *trace de dépliage* de G.

Exemple 16 *La trace de dépliage du GST de la Figure 6.3 est représentée sur la Figure 6.4.*

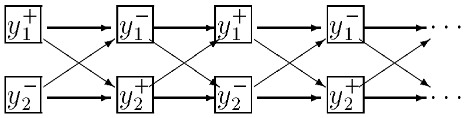

FIG. 6.4 – *Trace de dépliage d'un GST*

La trace de dépliage d'un GST présente des propriétés très particulières. Par exemple, on montre le résultat suivant, qui appartient au "folklore" de la théorie des GST :

Lemme 10 *Soit $G = (N, Y, \phi, \mathcal{V}_0)$ un GST avec $N = (P, T, F, M_{in})$, et soit TR_G la trace de dépliage de G. Alors :*

- TR_G *est périodique, i.e. il existe une trace finie u telle que* $\mathrm{TR}_G = u^\omega$.

- *On peut choisir u de sorte que $|u| = |T|$ et telle que pour tout $t \in T$, $|u|_t = 1$, i.e. chaque transition apparaît exactement une fois dans u.*

Preuve. [Schéma] Le réseau étant vivace, il existe une trace finie u sur T telle que u soit un comportement possible du système à partir de M_{in} et telle que toute transition t de T apparaisse dans u. Prenons u minimale, et considérons une linéarisation $t_1t_2\ldots t_n$ de u. On va alors montrer que pour tout $t \in T$, $|u|_t = 1$, i.e. chaque transition apparaît exactement une fois dans u.

Pour cela on raisonne par induction sur n. Il est clair que $|u|_{t_n} = 1$, sans quoi u ne serait pas minimal car $t_1 \ldots t_{n-1}$ possèderait la même propriété. Supposons maintenant que l'on ait montré que $|u|_{t_j} = 1$ pour tout $j > k$ et montrons que c'est encore le cas pour t_k.

Si t_k commute avec toutes les t_j, pour $j > k$, alors t_k est maximal dans u, donc il existe une linéarisation v' de u telle que t_k en soit la dernière lettre. Alors $|u|_{t_k} = 1$ sinon u ne serait pas minimal (même raisonnement que précedemment).

Sinon, il existe une transition t_j, avec $j > k$, qui dépend de t_k. Si t_k apparaissait au moins deux fois dans v, alors il existerait un $l < k$ tel que $t_l = t_k$ ($l > k$ est exclu par hypothèse d'induction). Or, entre deux occurrences successives d'une transition t, il y a nécessairement une occurrence de chacune des transitions t' dépendantes de t. Cela implique qu'il existe m tel que $l < m < k$ et $t_m = t_j$, ce qui contredit l'hypothèse d'induction $|u|_{t_j} = 1$. Donc $u_{t_k} = 1$.

On a donc montré qu'il existe une trace u telle que $\forall t \in T$, $|u|_t = 1$ exécutable à partir de M_{in}. On vérifie aisément que, lorsqu'on a exécuté une fois (et une seule) chacune des transitions, alors le système revient à son marquage d'origine : en effet, chaque place p du système ne peut être "vidée" que par une unique transition et ne peut être "remplie" que par une unique transition. On en conclut par induction que la trace u^ω est un comportement possible du système. Comme le système est déterministe (au sens où deux transitions franchissables sont toujours indépendantes) et l'alphabet de dépendance connexe, la trace u^ω est l'*unique* comportement infini du système, donc $\mathrm{TR}_G = u^\omega$. □

Remarque 10 *Le fait que $M_u = M_{in}$ n'implique pas que tous les marquages atteignables puissent être atteints en exécutant un préfixe de u. Considérons*

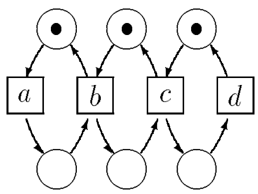

FIG. 6.5 – *Un exemple de GST*

par exemple le GST de la Figure 6.5, dont la trace de dépliage est représentée sur la Figure 6.6. Alors le marquage atteint lors de la configuration en pointillés n'est atteignable par aucun des préfixes de u. Nous reviendrons sur ce problème à la Section 6.5.

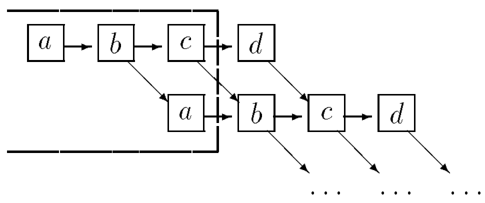

FIG. 6.6 – *Trace de dépliage*

La trace de dépliage TR_G d'un GST présente également une propriété essentielle :

Proposition 12 *Pour toute suite $t_1, t_2, \dots t_k$ de transitions franchissables à partir de M_{in}, i.e. telle que*

$$M_{in} \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \cdots \xrightarrow{t_k} M_k,$$

la trace $t_1 t_2 \dots t_k$ est un préfixe de u^ω.

Preuve. On commence par montrer que t_1 est un préfixe de u. Pour cela, on va montrer que t_1 est minimale dans u. En effet, si ce n'est pas le cas, alors il existe une transition t qui dépend de t_1 et qui apparaît avant t_1 dans u. Comme $(t, t_1) \in D_N$, il existe une place p qui est soit dans $t_1^\bullet \cap {}^\bullet t$, soit dans $t^\bullet \cap {}^\bullet t_1$. Supposons qu'elle soit dans $t^\bullet \cap {}^\bullet t_1$. Comme t_1 est franchissable en M_{in}, la place p est pleine pour ce marquage, et seule la transition t_1 peut la vider (car $|p^\bullet| = 1$). La transition t ne peut donc pas avoir lieu avant t_1 dans u car sinon le système ne serait plus 1-borné puisque t ajouterait un second jeton dans p. Un raisonnement du même type s'applique au cas où p appartient à $t_1^\bullet \cap {}^\bullet t$. On en conclut que t_1 est minimal dans u, donc dans u^ω. Posons $u = t_1 v$. Alors $u^\omega = t_1(v t_1)^\omega$ et on montre de la même façon que t_2 est minimale dans $v t_1$. De proche en proche, on en conclut que $t_1 t_2 \dots t_k$ est un préfixe de u^ω. □

Enfin, on montre la proposition suivante qui nous sera utile par la suite :

Proposition 13 *Soient c et c' deux configurations distinctes de* TR_G *de même marquage. Alors c et c' sont ordonnées et il existe un entier k tel que c et c' soient séparées par k occurrences de chacune des transitions t de T.*

Preuve. Soit M le marquage commun à c et à c'. Comme le système est déterministe, un même marquage implique un même futur, i.e. les suffixes de TR_G correspondants à c et à c' sont identiques. Comme l'ensemble des lettres infiniment répétées de TR_G forme un sous-alphabet *connexe* de (T, D_N), cela implique que c et c' sont ordonnées (par application du Lemme de Levi). Par exemple $c \subseteq c'$. On suppose les deux configurations distinctes. Soit v la trace telle que $c \xrightarrow{v} c'$. Considérons le système dont le marquage d'origine serait M. Alors on sait que tout comportement de ce système est un préfixe d'une trace w^ω, où w vérifie : pour tout $t \in T$, $|w|_t = 1$. Or, pour revenir au marquage M, il est nécessaire d'exécuter au moins w en raison de la connexité du système (car si une lettre t n'est pas exécutée alors toutes les lettres dépendantes de t ne le sont pas non plus, et par connexité aucune lettre ne l'est). On en conclut que $v = w^j$ pour un certain $j \geq 1$, donc que c et c' sont nécessairement séparées par j occurrences de chacune des transitions t de T. $\square$

6.4 Spécifications des GST

Nous allons maintenant utiliser la logique temporelle LTrL (cf. Chapitres 3 et 5) pour spécifier des propriétés dynamiques d'un graphe de signaux de transitions $G = (N, Y, \phi, \mathcal{V}_0)$, avec $N = (P, T, F, M_{in})$.

Pour cela, on commence par définir des opérateurs dérivés de ceux de LTrL sur l'alphabet (T, D_N). Pour tout $y \in Y$, soit $T_{y^\star} = \{t \mid \phi(t) = y^*\}$, où l'on rappelle que $\star \in \{+, -\}$. L'ensemble $T_{y^\star}$ désigne donc toutes les transitions "étiquetées" par $y^\star$. On définit alors, pour toute transition $t \in T$,

$$p_t \stackrel{\mathrm{def}}{=} \textcircled{t}\underline{tt}.$$

La propriété p_t est donc vérifiée par une configuration c de TR_G si la transition t est franchissable dans cette configuration. On définit aussi

$$p_{y^\star} \stackrel{\mathrm{def}}{=} \bigvee_{t \in T_{y^\star}} p_t.$$

La propriété $p_{y^\star}$ est vérifiée en c si une transition étiquetée (*via* ϕ) par $y^\star$ est franchissable dans cette configuration.

On définit aussi

$$[y^\star]\alpha \stackrel{\mathrm{def}}{=} \bigvee_{t \in T_{y^\star}} \textcircled{t}\alpha.$$

et les modalités $\Diamond$ et $\Box$ de la façon habituelle :

$$\Diamond\alpha \stackrel{\mathrm{def}}{=} \underline{tt}\ \mathcal{U}\ \alpha,$$

et

$$\Box\alpha \stackrel{\mathrm{def}}{=} \neg(\Diamond(\neg\alpha)).$$

Enfin, soit $T_y = T_{y^+} \cup T_{y^-}$. Considérons à présent la formule VLD^y suivante (pour VaLiDe):

$$\mathrm{VLD}^y \stackrel{\mathrm{def}}{=} \bigwedge_{\substack{t, t' \in T_y \\ t \neq t'}} \Box\neg(p_t \wedge p_{t'}).$$

Cette formule spécifie que deux transitions distinctes associées à un même signal y ne peuvent survenir concurremment. En effet, le franchissement simultané de deux transitions de type *up*, ou d'une transition *up* et d'une transition *down* sur le même signal serait révélateur d'une anomalie dans le système ou dans sa description.

Considérons maintenant la formule VLD_{y^+} suivante :

$$\mathrm{VLD}_{y^+} \stackrel{\mathrm{def}}{=} \neg\Diamond[y^+]((\neg p_{y^-})\,\mathcal{U}\,p_{y^+}).$$

Cette formule spécifie qu'il n'est pas possible pour une transition y^+ d'être suivie par une autre transition y^+ sans que soit intervenue une transition y^-. En fait, cette interprétation n'est valide que dans un contexte où la propriété précédemment définie VLD^y est déjà vérifiée. On définit la formule VLD_{y^-} d'une façon similaire.

Finalement, soit INIT^y la formule suivante :

$$\mathrm{INIT}^y \stackrel{\mathrm{def}}{=} \begin{cases} (\neg p_{y^-})\,\mathcal{U}\,p_{y^+} & \text{si } \mathcal{V}_0(y) = 0, \\ (\neg p_{y^+})\,\mathcal{U}\,p_{y^-} & \text{sinon.} \end{cases}$$

Cette formule spécifie que le premier signal de transition concernant y doit être y^+ si la valuation initiale de y est 0, et vice versa. Alors soit

$$\mathrm{VLD} = \bigwedge_{y \in Y} (\mathrm{VLD}^y \wedge \mathrm{VLD}_{y^+} \wedge \mathrm{VLD}_{y^-} \wedge \mathrm{INIT}^y).$$

Cette formule dit que toute séquence de transitions de G en partant du marquage initial M_{in} est *valide* au sens de [51], c'est-à-dire que la modélisation correspond bien à un système de transitions réel : pour tout y, la première transition concernant y est compatible avec la valuation initiale, et par la suite on a une alternance de transition *up* et de transitions *down* concernant cette variable, etc.

En plus de cette propriété de validité, il est aisé de définir toute une variété de propriétés de sûreté, de vivacité et de réponse en utilisant les opérateurs $\Box$, $\Diamond$ et $\mathcal{U}$.

6.5 Model Checking pour les GST

Après avoir donné un aperçu du type de propriétés que l'on peut écrire dans la logique LTrL à propos d'un GST, nous nous focalisons maintenant sur la validation de ces propriétés.

Soit $G = (N, Y, \phi, \mathcal{V}_0)$ un GST, avec $N = (P, T, F, M_{in})$, et soit α une formule de la logique LTrL sur l'alphabet (T, D_N) ne comportant pas de sous-formules de présent.

On a vu précédemment que tous les comportements finis de G sont des préfixes de la trace de dépliage TR_G. Cette trace représente donc l'unique comportement infini du système.

On dit que G *satisfait* la spécification α, et on note $G \models \alpha$, si $\mathrm{TR}_G, \emptyset \models \alpha$, i.e. si la trace de dépliage de G satisfait cette spécification. Le problème du model checking pour les GST est donc de décider, étant donnés G et α, si $G \models \alpha$. Dans ce cadre, notre résultat principal s'énonce comme suit :

Théorème 21 *Soit G et α définis comme ci-dessus. Alors la question $G \overset{?}{\models} \alpha$ peut être résolue en temps $O(|T|^2.|\mathrm{RM}_N|^2.|\alpha|)$, où $|T|$ est le nombre de transitions du réseau N sous-jacent à G, $|\mathrm{RM}_N|$ est le nombre de marquages atteignables de N, et $|\alpha|$ est la taille de la spécification.*

Puisque G ne possède qu'un *unique* comportement infini, représenté par la trace infinie TR_G, on peut espérer développer une technique similaire à celle employée pour la logique CTL sur les mots (*Computational Tree Logic*), c'est-à-dire une technique d'étiquetage inductif : on cherche à étiqueter chaque configuration de TR_G par les sous-formules de α qu'elle satisfait; on suppose que l'étiquetage est déjà effectué pour les sous-formules propres de α, et on développe un algorithme permettant alors d'étiqueter les configurations pour la formule α. Il s'agit donc d'une induction sur la structure de la formule α. Evidemment, la première difficulté à laquelle on se heurte est que la trace TR_G est infinie, donc comporte un nombre infini de configurations. Nous allons voir que dans le cas que nous étudions, cette difficulté est contournable.

Posons $\mathrm{TR}_G = (E, \leq, \lambda)$ et soit C_G l'ensemble des configurations de TR_G. On définit une application $m : C_G \to \mathrm{RM}_N$ de la façon suivante :

$$\forall c \in C_G, m(c) = M \text{ ssi } M_{in} \xrightarrow{c} M \text{ dans } \mathrm{RM}_N.$$

Alors l'application m satisfait $m(\emptyset) = M_{in}$ et pour tout couple $(c, c') \in C_G$, et toute transition $t \in T$,

$$c \xrightarrow{t} c' \text{ dans } \mathrm{TR}_G \text{ ssi } m(c) \xrightarrow{t} m(c') \text{ dans } \mathrm{RM}_N.$$

Nous allons maintenant montrer que l'on peut effectuer l'étiquetage précédemment évoqué sur un préfixe *fini* de la trace TR_G, grâce au Lemme suivant :

Lemme 11 *Il existe une constante effectivement calculable $d \leq |T|$ et des sous-ensembles $E_1, E_2 \subseteq E$ (on rappelle que $\mathrm{TR}_G = (E, \leq, \lambda)$) tels que les conditions suivantes soient satisfaites :*

1. *$E_1 \subseteq E_2$ et $E_1, E_2 \in C_G$. De plus, $|C_1| \leq d.|\mathrm{RM}_N|$ et $|C_2| \leq d.|\mathrm{RM}_N|$ où $C_1 = \{c \in C_G \mid c \subseteq E_1\}$ et $C_2 = \{c \in C_G \mid E_1 \subseteq c \subseteq E_2\}$. Enfin, $m(E_1) = m(E_2) = M_{in}$.*

2. *Si $M \in \mathrm{RM}_N$ alors il existe $c_1 \in C_1$ et $c_2 \in C_2$ tels que $m(c_1) = m(c_2) = M$.*

3. *Si $M, M' \in \mathrm{RM}_N$ de sorte que M' soit atteignable à partir de M alors il existe $c, c' \in C_1 \cup C_2$ avec $c \subseteq c'$ tels que $m(c) = M$ et $m(c') = M'$.*

Preuve. Soit u la trace finie définie par le Lemme 10. On rappelle que $\mathrm{TR}_G = u^\omega$ et que u contient une occurrence de chaque transition de T. Sur le graphe de l'alphabet de dépendance (T, D_N), on définit une distance comme suit : pour tout couple $(t, t') \in T \times T$, $d(t, t')$ est la longueur du plus court chemin menant de t à t' dans (T, D_N). L'alphabet étant connexe, cette distance existe toujours. On pose alors

$$d = 1 + max_{t,t' \in T}\ d(t, t').$$

On a alors trivialement $d \leq |T|$.

Considérons à présent la trace u^d. Ce préfixe de TR_G correspond à une configuration que l'on note E_1. De même, on note E_2 la configuration liée au préfixe u^{2d} de TR_G. On a ainsi $E_1 \subseteq E_2$. Soient C_1 et C_2 définies comme dans l'énoncé du lemme, en fonction de E_1 et E_2. On a déjà vu que $m(u) = M_{in}$. Il est donc clair que $m(u^d) = m(u^{2d}) = M_{in}$, i.e. $m(E_1) = m(E_2) = M_{in}$.

On va maintenant montrer que tous les marquages accessibles $M \in \mathrm{RM}_N$ peuvent être atteint via une configuration de C_1. Soit M un tel marquage, et soit c une configuration de TR_G telle que :

- $m(c) = M$.

- Pour toute configuration $c' \in C_G$, si $m(c') = M$ alors $c \subseteq c'$.

Si $c \in C_1$, il n'y a rien à faire. Sinon, cela signifie qu'il existe au moins un sommet $e \in c$ tel que $e \notin E_1$. On va montrer à présent que tout sommet e' appartenant à c_u, la configuration de TR_G correspondante au préfixe u, est antérieure à e. Soit un tel $e' \in c_u$. Posons $a_0 = \lambda(e')$ et $a_d = \lambda(e)$. D'après la définition de d, il existe un chemin $a_0, a_1, \ldots, a_d$ dans le graphe (T, D_N) de a_0 vers a_d, en autorisant des répétitions éventuelles (i.e. par exemple $a_2 = a_3$). Pour tout i tel que $0 < i < d$, soit e_i *le* sommet d'étiquette a_i de l'ensemble $c_{u^{i+1}} \setminus c_{u^i}$, où c_v désigne la configuration de TR_G correspondant au préfixe v de TR_G. En posant $e_0 = e'$ et $e_d = e$, on a alors :

$$e_0 \leq e_1 \leq e_2 \leq \ldots \leq e_d,$$

et par conséquent $e' \leq e$. Ceci étant vrai pour tout $e' \in c_u$, et sachant que $\downarrow\!e \subseteq c$, on en déduit que $c_u \subseteq c$. La trace u est donc un préfixe de la trace v associée à c. Posons $v = uw$, et soit c' la configuration associée à w. Comme $m(c_u) = M_{in}$, on a nécessairement $m(c') = m(c)$. Or c' est strictement plus petite que c, ce qui contredit la minimalité de c. On en conclut que tout événement de c est dans E_1, donc que $c \in C_1$. Tous les marquages accessibles le sont donc via une configuration de C_1.

On va maintenant chercher à majorer le nombre de configurations qui sont dans C_1. Supposons que $c, c' \in C_1$, avec $m(c) = m(c')$, les configurations c et c' étant distinctes. Grâce à la Proposition 13, on sait que ces deux configurations sont séparées par $k.|T|$ événements, où k est un entier. Autrement dit, pour un marquage donné, et sachant que E_1 correspond à la trace u^d (de taille $d.|T|$), il y a au plus d configurations de C_1 correspondantes à ce marquage. D'où l'inégalité

$$|C_1| \leq d.|\mathrm{RM}_N|.$$

De la même façon, on montre que

$$|C_2| \leq d.|\mathrm{RM}_N|.$$

Soient M et M' deux marquages atteignables tels que M' soit atteignable à partir de M. On sait qu'il existe $c_1 \in C_1$ telle que $m(c_1) = M$. Il existe aussi $c_2 \in C_2$ telle que $m(c_2) = M'$. Or on a nécessairement $c_1 \subseteq E_1 \subseteq c_2$. □

Grâce à ce lemme, on va pouvoir travailler sur un préfixe fini de TR_G, et donc écrire un algorithme d'étiquetage efficace. Soit $\mathrm{TR}_{\mathrm{fin}}$ le préfixe de TR_G correspondant à la configuration E_2. Soit C l'ensemble des configurations de

$\mathrm{TR}_{\mathrm{fin}}$. Alors C possède au plus $2d|\mathrm{RM}_N|$ éléments.

Soit α_0 une formule de la logique temporelle LTrL sur l'alphabet de dépendance (T, D_N), construite inductivement à partir des formules du type p_t, des opérateurs $[t]$ et $\mathcal{U}$ et des connecteurs booléens. On exclut ainsi les formules de présent. Nous allons étiqueter inductivement les membres de C par les sous-formules de la spécification α_0. Nous affirmons que $G \models \alpha_0$ ssi *in fine* la configuration vide est étiquetée par α_0.

Remarque 11 *Soient $c, c' \in C$. Si c et c' possèdent le même marquage, alors elles satisfont les mêmes formules. En effet, $m(c) = m(c')$ implique que les suffixes de TR_G correspondant à c et à c' sont identiques (puisque le système est déterministe). Ces deux configurations "voient" donc le même futur, et donc satisfont les mêmes propriétés (tant que l'on utilise pas de formules de présent).*

Algorithme d'étiquetage : on distingue différents cas suivant la structure de la formule α_0. On note Λ la fonction qui associe à tout élément de $C_1 \cup C_2$ un ensemble de sous-formules de α_0. Initialement, on pose $\Lambda(c) = \{\underline{tt}\}$ pour tout $c \in C$.

- Si $\alpha_0 = p_t$, on étiquette trivialement les membres de C : pour tout $c \in C$, on ajoute p_t à $\Lambda(c)$ ssi t est franchissable en c.
- Les cas $\alpha_0 = \alpha \vee \beta$ et $\alpha_0 = \neg\alpha$ ne présentent pas de difficultés et sont traités de la façon usuelle : par exemple, on étiquette par $\alpha \vee \beta$ toutes les configurations qui sont déjà étiquetées soit par α soit par β.
- Si $\alpha_0 = [t]\alpha$, on ajoute $[t]\alpha$ à $\Lambda(c)$ ssi t est franchissable en c et si son franchissement conduit à une configuration $c' \in C$ telle que $\alpha \in \Lambda(c')$.
- Si $\alpha_0 = \alpha\, \mathcal{U}\, \beta$, remarquons dans un premier temps que l'on peut coder tout événement de E_2 (donc de $\mathrm{TR}_{\mathrm{fin}}$) par une paire (t, n) unique, où $t = \lambda(e)$ est l'étiquette de e et $n = |\downarrow e|_t$ est le nombre d'occurrences de t dans le passé de e (on a clairement $n \leq 2d$). Ce codage est donc fini. Chaque configuration c de C peut être décrite par l'ensemble de ses éléments maximaux, et ainsi peut être codée par un ensemble unique et fini $s(c)$ tel que :

$$s(c) = \{(t, n) \mid (t, n) \text{ code un élément maximal de } c\}.$$

Ainsi, chaque configuration du préfixe fini $\mathrm{TR}_{\mathrm{fin}}$ est codée de façon unique et finie. De plus, le calcul de n pour un événement e donné est aisé, puisque les occurrences de t sont totalement ordonnées. Ceci étant dit, on identifira par la suite les configurations avec leur codage, afin d'alléger l'écriture de l'algorithme.

Dans le préfixe fini $\mathrm{TR}_{\mathrm{fin}}$, on commence par étiqueter les configurations les plus grandes et on procède ensuite à l'étiquetage des configurations plus petites, niveau par niveau, où le NIVEAU d'une configuration est égal à sa taille moins celle de E_2[1]. Le NIVEAU maximal est donc 0. En d'autres termes, on travaille d'abord sur les configurations de taille m avant de considérer les configurations de taille $m-1$. Notre algorithme utilise un ensemble global COL qui sert à "collecter" toutes les configurations pour lesquelles β est satisfaite. De plus, à toute configuration c, nous allons associer l'ensemble $\mathrm{col}(c)$ des configurations de COL qui sont atteignables à partir de c (i.e. qui contiennent c). Enfin, pour toute configuration x de $\mathrm{col}(c)$, on va calculer un prédicat booléen $\mathrm{statut}(c,x)$ qui va indiquer si $\alpha\ \mathcal{U}\ \beta$ est satisfaite en c avec x comme témoin (i.e. $x \models \beta$, $c \subseteq x$, et pour tout c', $c \subseteq c' \subset x$ implique $c'' \models \alpha$). Si c satisfait $\alpha\ \mathcal{U}\ \beta$ avec x pour témoin, on note $c \models_x \alpha\ \mathcal{U}\ \beta$. On montre alors aisément que $c \models_x \alpha\ \mathcal{U}\ \beta$ ssi

- $x = c$ et $c \models \beta$
  ou
- $c \models \alpha$ et pour tout c' telle que $c \to c' \xrightarrow{\star} x$ on a $c' \models_x \alpha\ \mathcal{U}\ \beta$.

Cette remarque est au coeur de notre algorithme, qui procède comme suit :

1. Posons COL $= \emptyset$ et NIVEAU $= 0$. Remarquons qu'il n'existe qu'une seule configuration de niveau 0 (il s'agit de E_2) et que le marquage correspondant à cette configuration est le marquage initial M_{in}.
2. Pour toute configuration c du niveau courant :
   (a) Si c satisfait β (i.e. $\beta \in \Lambda(c)$), on ajoute c à COL et à $\mathrm{col}(c)$, et on pose $\mathrm{statut}(c,c) = 1$ (l'ensemble $\mathrm{col}(c)$ est initialement vide).

1. la taille d'une configuration étant simplement son nombre de sommets.

(b) Pour toute configuration x dans COL telle que c possède un successeur c' vérifiant $x \in \mathrm{col}(c')$, on ajoute x à $\mathrm{col}(c)$.

(c) Pour toute configuration x dans $\mathrm{col}(c)$, si une des deux conditions suivantes est satisfaite :
   - c ne satisfait pas α (i.e. $\alpha \notin \Lambda(c)$);
   - c a un successeur c' vérifiant $x \in \mathrm{col}(c')$ et $\mathrm{statut}(c', x) = 0$;

   alors on pose $statut(c, x) = 0$. Sinon, on pose $statut(c, x) = 1$.

(d) S'il existe x dans $\mathrm{col}(c)$ telle que $\mathrm{statut}(c, x) = 1$, alors on ajoute $\alpha\, \mathcal{U}\, \beta$ à $\Lambda(c)$.

3. On passe au niveau inférieur : NIVEAU : $=$ NIVEAU $- 1$ et on recommence.
4. Quand on est arrivé au niveau minimal, on reporte les étiquetages des configurations de C_1 vers celles de $C \setminus C_1$ de la façon suivante : pour tout $c \in C \setminus C_1$, il existe une configuration c' de C_1 qui possède le même marquage que c; on ajoute $\alpha\ \mathcal{U}\ \beta$ à $\Lambda(c)$ ssi $\alpha\, \mathcal{U}\, \beta \in \Lambda(c')$.

Théorème 22 *L'algorithme d'étiquetage est valide, au sens où pour toute configuration c de C, on a :*

$$c \models \alpha_0 \ ssi \ \alpha_0 \in \Lambda(c).$$

Preuve. On procède par induction sur la structure de la formule α_0. Il est clair que le seul cas difficile est celui où α_0 est de la forme $\alpha\, \mathcal{U}\, \beta$, aussi nous nous focalisons sur celui-ci. Par induction, on suppose que l'étiquetage est valide pour les sous-formules α et β de α_0. Comme l'algorithme fonctionne “à l'envers”, si deux configurations c et c' sont telles que $c \subseteq c'$, alors c' sera visitée avant c. Par conséquent, lorsqu'une configuration c est visitée, toutes les configurations qui sont dans le futur de c l'ont déjà été. Ceci étant, il est clair que l'étape $2(a)$ de l'algorithme assure que toutes les configurations satisfaisant β vont être ajoutées dans l'ensemble COL, et celles-là uniquement. Ainsi, lorsqu'une configuration c est visitée, toutes les configurations plus grandes que c qui satisfont β sont déjà dans COL. De plus, l'étape $2(b)$ de l'algorithme fait en sorte que toutes les configurations plus grandes que c et satisfaisant β se trouvent dans $\mathrm{col}(c)$. Le cas particulier où c elle-même satisfait β est traité en $2(a)$. Enfin, l'étape $2(c)$ assure que $\text{statut}(c,x) = 1$ ssi les deux conditions suivantes sont satisfaites :

- c satisfait α;
- pour tout successeur c' de c, si $x \in \text{col}(c')$ alors $\text{statut}(c',x) = 1$.

Ainsi seules les configurations x plus grandes que c, satisfaisant β et telles que α soit satisfaite entre c et x vérifient $\text{statut}(c,x) = 1$. Ceci équivaut bien au fait que x soit un témoin pour c de $\alpha\,\mathcal{U}\,\beta$. L'étape $2(d)$ se contente alors d'ajouter $\alpha\,\mathcal{U}\,\beta$ à $\Lambda(c)$ ssi c possède un témoin pour $\alpha\,\mathcal{U}\,\beta$. Afin de conclure à la validité de l'algorithme, il reste à remarquer que, si une configuration c de C_1 satisfait $\alpha\ \mathcal{U}\ \beta$, et si on note c' une configuration témoin pour c alors il existe dans C_2 une configuration c_t qui possède le même marquage que c', qui est dans le futur de c, et qui par conséquent est également une configuration témoin pour c. Si maintenant on considère une configuration c de $C \setminus C_1$ qui satisfait $\alpha\,\mathcal{U}\,\beta$, alors il existe dans C_1 une configuration c' de même marquage que c, et donc c' satisfait également $\alpha\,\mathcal{U}\,\beta$. Par conséquent, c' sera étiquetée par $\alpha\,\mathcal{U}\,\beta$ par l'algorithme au cours des étapes 1 à 3, et c le sera également au plus tard au cours de l'étape 4. □

Théorème 23 *L'algorithme d'étiquetage précédemment décrit possède une complexité en temps* $O(|T|^2|\text{RM}_N|^2|\alpha_0|)$.

Preuve. Il est clair que les formules du type $\alpha\ \mathcal{U}\ \beta$ sont celles qui demandent le plus de temps pour l'algorithme d'étiquetage. Plaçons nous dans le cas d'une telle formule. Comme il y a au plus $2d|\text{RM}_N|$ configurations dans $C_1 \cup C_2$, l'algorithme va passer au plus $2d|\text{RM}_N|$ fois par les étapes $2(a)$ à $2(d)$. Chacune de ces sous-étapes peut nécessiter jusqu'à $O(2d|\text{RM}_N|)$ unités de temps pour être exécutées, car les ensembles COL et $\text{col}(c)$ peuvent contenir au plus $2d|\text{RM}_N|$ éléments. Le temps d'étiquetage pour une formule du type *until* (en supposant l'étiquetage des sous-formules déjà effectué) est donc en $O(d^2|\text{RM}_N|^2)$, c'est-à-dire, puisque $d \leq |T|$, en $O(|T|^2|\text{RM}_N|^2)$. Par induction, si $|\alpha_0|$ est la taille de la formule dont on veut effectuer l'étiquetage, alors l'ensemble du processus se fait en temps $O(|T|^2|\text{RM}_N|^2|\alpha_0|)$. □

La conjonction des Théorèmes 22 et 23 implique trivialement le Théorème 21 : il suffit de faire exécuter l'algorithme d'étiquetage pour la formule α et de vérifier ensuite si la configuration initiale $\emptyset$ satisfait $\alpha \in \Lambda(\emptyset)$.

6.6 Conclusion

On a montré dans ce chapitre que, malgré le caractère non-élémentaire de la logique LTrL, on pouvait développer un algorithme de model-checking efficace dans le cadre restreint de la vérification des graphes de signaux de transitions, car alors il n'y a qu'un comportement possible, et la trace qui représente ce comportement possède des régularités très fortes. On peut donc se demander si un tel algorithme est généralisable à une classe plus grande de graphes. En particulier, on pourrait s'intéresser dans un premier temps aux réseaux de Petri appelés *schémas de synchronisation bipolaires* [16], qui forment une classe un peu plus vaste que les GST, et qui permettent dans une certaine mesure de modéliser des entrées, ce que n'autorisaient pas les GST.

Chapitre 7

Conclusion et perspectives

Résultats Dans cette thèse nous avons apporté plusieurs résultats concernant les logiques temporelles sur les traces, et en particulier :

- au chapitre 4, nous avons montré un résultat de séparabilité des formules de la logique temporelle locale TrPTL dans le cadre d'un alphabet de dépendance non-connexe (chapitre 4). Ce résultat est à rapprocher du Théorème de Mezei [2] sur les mots.

- au chapitre 5, nous avons exhibé un algorithme de décision modulaire pour la logique temporelle globale et non-élémentaire LTrL. Cet algorithme apporte une solution directe au problème de la satisfaisabilité des formules de cette logique.

- au chapitre 6, nous avons abordé un aspect plus pratique de l'utilisation de LTrL, en proposant un algorithme de model-checking de faible complexité dans le cadre restreint des graphes de signaux de transitions, un formalisme utilisé pour spécifier des architectures matérielles.

Perspectives Plusieurs perspectives de recherche se présentent naturellement à nous :

- la recherche de la logique temporelle idéale pour les traces. Rappelons qu'il s'agit d'exhiber une logique temporelle qui soit à la fois élémentaire (et si possible de complexité raisonnable) et expressivement complète (i.e. dont le pouvoir d'expression est celui de la logique du premier ordre). L'existence d'une telle logique est aujourd'hui un problème ouvert. On est donc amené soit à exhiber une telle logique, soit à démontrer sa non-existence. Dans un cas comme dans l'autre, plusieurs voies peuvent nous permettre d'avancer dans la compréhension de ce problème. Une approche possible est de chercher à expliciter plus clairement les liens entre les différentes logiques existantes et en particulier à comprendre les puissances relatives des différents opérateurs temporels qui y sont utilisés. On voit par exemple que l'utilisation d'une logique locale s'avère plus adaptée à la spécification d'un processus particulier au sein d'un système distribué en comportant plusieurs, alors que l'utilisation d'une logique globale convient mieux à la spécification de l'ensemble du système. Néanmoins, les liens entre ces deux types de logiques temporelles sont loin d'être élucidés. Au chapitre 4 nous avons apporté une contribution à ce problème en écrivant un opérateur global *via* des opérateurs locaux. La taille de la formule résultante montre que ce genre de traduction n'est a priori pas une chose facile.

- l'isolement de sous-classes de modèles dans lesquels les problèmes de la vérification présentent des complexités faibles pour les logiques temporelles existantes. Les résultats du chapitre 6 ouvrent une voie dans ce cadre en montrant que l'utilisation d'une logique comme LTrL peut s'avérer efficace dans la pratique. Il s'agit alors de chercher à généraliser peu à peu la classe considérée tout en conservant une complexité faible pour le model-checking.

Bibliographie

[1] R. Alur, K. McMillan, and D. Peled. Deciding global partial-order properties. In *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 1998.

[2] J. Berstel. *Transductions and context-free languages.* Teubner Studienbücher, 1979.

[3] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science.* Stanford Univ. Press, 1962.

[4] J. Cohen, D. Perrin, and J. E. Pin. On the expressive power of temporal logic. *Journal of Computer and System Sciences*, 46:271–295, 1993.

[5] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[6] R. Cori and D. Perrin. Automates et commutations partielles. *R.A.I.R.O. — Informatique Théorique et Applications*, 19:21–32, 1985.

[7] V. Diekert and P. Gastin. An expressively complete temporal logic without past tense operators for Mazurkiewicz traces. In *Proceedings of CSL'99*, à paraître.

[8] V. Diekert and Y. Métivier. Partial commutation and traces. In *Handbook of Formal Languages*, pages 457–533. Springer, 1996.

[9] V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.

[10] W. Ebinger. *Charakterisierung von Sprachklassen unendlicher Spuren durch Logiken.* Dissertation, Institut für Informatik, Universität Stuttgart, 1994.

[11] W. Ebinger and A. Muscholl. Logical definability on infinite traces. In *Proceedings of ICALP'93*, volume 700 of *Lecture Notes in Computer Science*, pages 335–346. Springer, 1993.

[12] P. Gastin, R. Meyer, and A. Petit. A (non-elementary) modular decision procedure for LTrL. In *Proceedings of MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 356–365. Springer, 1998.

[13] P. Gastin and A. Petit. Asynchronous automata for infinite traces. In *Proceedings of ICALP'92*, volume 623 of *Lecture Notes in Computer Science*, pages 583–594. Springer, 1992.

[14] P. Gastin, A. Petit, and W. Zielonka. A Kleene theorem for infinite trace languages. In *Proceedings of ICALP'91*, volume 510 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 1991.

[15] P. Gastin, A. Petit, and W. Zielonka. An extension of Kleene's and Ochmański's theorems to infinite traces. *Theoretical Computer Science*, 125:167–204, 1994.

[16] H.J. Genrich and P.S. Thiagarajan. A theory of bipolar synchronization schemes. *Theoretical Computer Sience*, 30:241–318, 1984.

[17] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of LICS'91*, pages 406–415. IEEE Computer Society Press, 1991.

[19] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110:305–326, 1994.

[20] G. Guaiana, A. Restivo, and S. Salemi. On aperiodic trace languages. In *Proceedings of STACS'91*, volume 480 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 1991.

[21] H. Kamp. *Tense Logic and the Theory of Linear Order.* Phd thesis, University of California, 1968.

[22] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75(3):21–43, 1991.

[23] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[24] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Proceedings of CSL'97*, volume 1414 of *Lecture Notes in Computer Science.* Springer, 1997.

[25] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev. Identifying state coding conflicts in asynchronous system specifications using petri net unfoldings. In *International Conference on Application of Concurrency to System Design*, pages 152–163, Japan, March 1998.

[26] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, 1998.

[27] D. Kuske. On temporal logics for traces with quantification. Technical report, Teschnische Universität Dresden, December 1996.

[28] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[29] K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 164–174. Springer, 1992.

[30] R. McNaughton and S. Papert. *Counter-free Automata.* MIT Press, 1971.

[31] R. Meyer and A. Petit. Decomposition of TrPTL formulas. In *Proceedings of MFCS'97*, number 1295 in Lecture Notes in Computer Science, pages 418–427. Springer, 1997.

[32] R. Meyer and P.S. Thiagarajan. LTrL-based model-checking for a restricted class of Signal Transition Graphs. In *Hardware Design and Petri Nets*. Kluwer Academic Publishers, à paraître.

[33] M. Mukund and P.S. Thiagarajan. Linear time temporal logics over Mazurkiewicz traces. In *Proceedings of MFCS'96*, volume 1113 of *Lecture Notes in Computer Science*, pages 62–92. Springer, 1996.

[34] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, Chicago, Illinois, 28–30 October 1963. IEEE.

[35] Edward Ochmański. *Regular trace languages (in Polish)*. PhD thesis, Warszawa, 1984.

[36] W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California Los Angeles, 1981.

[37] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[38] D. Peled. Partial-order reductions: model checking using representatives. In *Proceedings of MFCS'96*, volume 1113 of *Lecture Notes in Computer Science*, pages 93–112. Springer, 1996.

[39] A. Pnueli. The temporal logics of programs. In *Proceedings of FOCS'77*, pages 46–57, 1977.

[40] R. Ramanujam. Locally linear time temporal logic. In *Proceedings of LICS'96*, pages 118–127. IEEE Computer Society Press, 1996.

[41] L.Y. Rosenblum and A.V.Yakolev. Signal graphs : from self-timed to timed ones. In *Proc. of International workshop on Timed Petri nets*, pages 199–207. IEEE Computer Society Press, 1985.

[42] S. Safra. On the complexity of ω-automata. In *Proceedings of the 29th annual IEEE Symp. on Foundations of Computer Science*, pages 319–327, 1988.

[43] A. Sistla and E. Clarke. The complexity of propositional linear time logic. *Journal of the ACM*, 32:733–749, 1985.

[44] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proceedings of LICS'94*, pages 438–447. IEEE Computer Society Press, 1994.

[45] P. S. Thiagarajan. A trace consistent subset of PTL. In *Proceedings of CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 1995.

[46] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Proceedings of LICS'97*, pages 183–194. IEEE Computer Society Press, 1997.

[47] A. Valmari. *State Space Generation: Efficiency and Practicality.* PhD thesis, Tampere University of Technology, 1988.

[48] A. Valmari. A Stubborn Attack on the State Explosion Problem. In *Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 1990.

[49] A. Valmari. A stubborn attack on state explosion. In *Formal Methods in System Design*, pages 1:297–322, 1992.

[50] I. Walukiewicz. Difficult configurations - on the complexity of LTrL. In *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 1998.

[51] A. V. Yakovlev and A. M. Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets I*, volume 1491 of *Lecture Notes in Computer Science*, pages 154–236. Springer, 1998.

[52] L. Zuck. *Past Temporal Logic.* PhD thesis, Weizmann Institute, Rehovot, Israel, 1986.