Université Paris 7 - Denis Diderot
UFR d'Informatique

THÈSE

pour l'obtention du titre de
Docteur de l'Université Paris 7
Spécialité Informatique

présentée par

Denis Oddoux

sur le sujet

Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéraires

soutenue publiquement le 17 décembre 2003 devant le jury suivant :

Jean-Éric Pin	Président du Jury
Béatrice Bérard	Rapporteuse
Nicolas Halbwachs	Rapporteur
Paul Gastin	Directeur de Thèse
Thomas Wilke	
Pierre Wolper	

Remerciements

Je remercie Jean-Éric Pin de m'avoir fait l'honneur de présider le jury de cette thèse, Béatrice Bérard et Nicolas Halbwachs d'avoir accepté d'en être les rapporteurs, Thomas Wilke et Pierre Wolper pour avoir accepté de participer au jury.

Je tiens particulièrement à remercier Paul Gastin pour m'avoir guidé dans mon travail au cours de ces années, pour son attention et son aide à mes travaux de recherche, pour sa disponibilité exceptionnelle, et pour avoir été capable, en toutes circonstances, de trouver une solution constructive à chacune de mes difficultés.

Je remercie enfin les joyeux lurons de PPS pour leur accueil et leur soutien moral au cours des nombres pauses-goûter autour de la fontaine, ainsi que l'ensemble des personnes que j'ai pu rencontrer au cours de ces trois années au LIAFA.

Table des matières

Table des matières **5**

1 Introduction générale **9**
1.1 Les méthodes de vérification . . . 9
1.1.1 Les tests . . . 9
1.1.2 Les méthodes formelles . . . 10
1.1.3 Méthodes formelles ou tests, que choisir ? . . . 11
1.2 Le model-checking . . . 11
1.2.1 Le modèle . . . 12
1.2.2 La spécification . . . 12
1.2.3 Les algorithmes de model-checking . . . 13
1.3 Contenu de la thèse . . . 13

I Notions de base **15**

2 Logiques temporelles linéaires **17**
2.1 LTL . . . 18
2.1.1 Définitions . . . 18
2.1.2 Simplification . . . 19
2.2 PLTL : LTL avec passé . . . 20
2.3 Exemples . . . 22

3 Model-checking pour LTL et PLTL **23**
3.1 Formalismes utilisés . . . 23
3.2 Algorithme général de model-checking . . . 24
3.2.1 Complexité . . . 24
3.2.2 Principe général . . . 25
3.2.3 Explications . . . 26
3.3 De la logique temporelle aux automates . . . 27
3.3.1 La méthode des tableaux . . . 27
3.3.2 Méthodes incrémentales . . . 29

II Model-checking efficace pour LTL 31
Introduction 33
Motivations 33
Approche utilisée 33
Simplifications 34
Implémentation 34
Plan de la partie 34

4 Automates alternants 37
4.1 Définitions 37
4.1.1 Combinaisons booléennes positives 37
4.1.2 Automates alternants 38
4.1.3 Q-forêt, exécution, langage 38
4.1.4 Q-DAGs 40
4.2 Exemple 42
4.3 De LTL aux automates alternants très faibles 44
4.3.1 Construction 44
4.3.2 Preuve de l'équivalence 45
4.3.3 Complexité 49

5 Automates de Büchi généralisés 51
5.1 Définition 51
5.2 Exemple 52
5.3 Des automates alternants très faibles aux automates de Büchi généralisés 52
5.3.1 Construction 52
5.3.2 Preuve de l'équivalence 54
5.3.3 Complexité 56

6 Model-checking pour LTL 57
6.1 Des automates de Büchi généralisés aux automates de Büchi classiques 57
6.1.1 Construction 58
6.1.2 Automates d'acceptation 58
6.1.3 Preuve de l'équivalence 60
6.1.4 Complexité 61
6.2 Automates de Büchi généralisés et model-checking 61
6.2.1 Principe 61
6.2.2 Complexité 61

7 Implémentation 63
7.1 Optimisation des structures de données 63
7.1.1 L'étiquetage des transitions 63
7.1.2 Transitions des automates alternants 64
7.1.3 Définitions et constructions 65

7.2 Simplification des automates 67
7.2.1 Simplification des transitions 68
7.2.2 Simplification des états 71
7.2.3 Utilisation des composantes fortement connexes 72
7.2.4 Simplification à la volée 73
7.3 Résultats expérimentaux 73
7.4 L'outil LTL2BA 76
7.4.1 Utilisation 76
7.4.2 Structure du programme 77
7.4.3 Interface web 78
7.4.4 Références web 78

III L'extension à PLTL 81

Introduction 83
Motivations 83
Choix d'une approche 83
Approche utilisée 84
Plan de la partie 85

8 Automates alternants à double sens 87

8.1 Définitions 87
8.1.1 Automates alternants à double sens 87
8.1.2 Automates progressants 89
8.2 Exemple 95
8.3 Des automates alternants très faibles à double sens aux automates progressants 96
8.3.1 Construction 96
8.3.2 Preuve de l'équivalence 97
8.3.3 Complexité 99
8.4 De PLTL aux automates progressants très faibles 99
8.4.1 Construction 99
8.4.2 Preuve de l'équivalence 101
8.4.3 Complexité 110

9 Automates de Büchi à mémoire 111

9.1 Définition 112
9.2 Des automates progressants très faibles aux automates de Büchi à mémoire 112
9.2.1 Première construction 112
9.2.2 Preuve de l'équivalence 113
9.2.3 Complexité 114
9.2.4 Deuxième Construction 114
9.2.5 Preuve de l'équivalence 115
9.2.6 Complexité 118

10 Model-checking pour PLTL **119**
10.1 Des automates de Büchi à mémoire aux automates de Büchi classiques . . 119
10.1.1 Construction 119
10.1.2 Complexité 120
10.2 Automates de Büchi à mémoire et model-checking 120

11 Implémentation **121**
11.1 Optimisation des structures de données 121
11.1.1 Étiquetage des transitions 121
11.1.2 Transitions des automates alternants 121
11.2 Simplification des automates 122
11.2.1 Simplification des transitions 122
11.2.2 Simplification des états 123
11.3 Construction de l'automate de Büchi à mémoire 124
11.4 Résultats expérimentaux 126
11.5 L'outil PLTL2BA 126

12 Conclusion et perspectives **129**
Bilan de cette thèse 129
LTL 129
PLTL 130
Travaux futurs 130

Index **131**

Bibliographie **133**

Chapitre 1

Introduction générale

1.1 Les méthodes de vérification

Toute personne ayant déjà manipulé des logiciels informatiques a déjà rencontré des défaillances, ou *bugs*. Ces défaillances sont en général causées par une erreur du concepteur du logiciel, qu'il n'a pas pu identifier malgré une ou plusieurs relectures attentives. Et pour cause, tout développeur sait que plus son programme devient gros, plus il est difficile de s'assurer que celui-ci possède le comportement souhaité dans toutes les situations et de le concevoir sans erreur.

1.1.1 Les tests

La première méthode utilisée pour traquer les défaillances est le test. Le développeur ou une autre personne utilise le programme et prend en compte les erreurs ou les incohérences qu'il rencontre, en essayant de prévoir tous les comportements que pourra avoir un utilisateur. Cette méthode porte ses fruits dans un premier temps, mais s'avère vite limitée, puisqu'il est difficile d'anticiper toutes les actions que peut effectuer un utilisateur, en particulier si on veut que l'application soit robuste aux actions d'une personne qui ne sait pas comment l'utiliser.

Une deuxième méthode qui apparaît alors naturellement est l'utilisation de tests aléatoires intensifs. Un test aléatoire prend en compte toutes les actions qui sont à la disposition d'un utilisateur, et les enchaîne de façon désordonnée tout en surveillant que le programme se comporte comme il le doit. Effectuer un grand nombre de tels tests en séquence permet en général de déceler de nouvelles erreurs, mais il est impossible de tester tous les scénarios, et la plupart du temps le programme comporte encore des *bugs* à l'issue de cette étape.

La part croissante que l'informatique occupe dans des domaines critiques comme les télécommunications, l'aviation, le nucléaire ou la médecine impose de pouvoir garantir de façon absolue l'absence de toute défaillance dans certains logiciels. C'est ce que proposent les *méthodes formelles*.

1.1.2 Les méthodes formelles

Il existe de trop nombreux cas historiques tristement célèbres où l'utilisation de méthodes formelles aurait permis d'isoler des erreurs que les simples tests et simulations n'ont pas pu déceler à temps.

En plus de l'exemple très classique aujourd'hui de l'explosion de la fusée *Ariane V* au cours de son premier lancement en 1996 [Ari96], de nombreux autres cas similaires se sont produits, comme la grande panne du réseau téléphonique américain en 1989. C'est d'ailleurs en conséquence de cette panne que les *Bell Labs* se sont intéressés au *model-checking*.

Les méthodes formelles s'appuient sur deux données : l'application informatique, qui est *modélisée* afin de pouvoir être manipulée efficacement, et une *spécification* qui exprime une ou plusieurs propriétés que l'application doit réaliser.

Les méthodes formelles les plus courantes sont les suivantes :

Les simulations : celles-ci sont le pendant des tests dans un cadre formel. Ainsi, plutôt que d'effectuer des tests sur un programme ou sur un matériel informatique, on peut les effectuer sur un modèle formel qui respecte à la lettre son comportement. Ceci permet, en particulier dans le cas d'un système matériel, d'éliminer les erreurs de conception avant la production du circuit, plutôt que de les déceler *a posteriori* et d'assumer le coût de production d'une maquette ou, pire, d'une série défectueuse.

Si cette méthode présente des avantages par rapport aux tests, elle présente aussi le même inconvénient : il n'existe à l'issue de cette vérification aucune garantie de l'absence de bugs.

Les preuves automatiques : il s'agit de prouver pas à pas que le programme satisfait la spécification, en prouvant des résultats intermédiaires au fur et à mesure comme lorsqu'on prouve un résultat en mathématiques. Lorsqu'un outil de preuve automatique assure que la spécification est vérifiée, alors le modèle ne peut pas contenir de défaillances. Cependant, il est impossible de construire un outil pouvant trouver une preuve pour tous les programmes corrects, puisque la logique mathématique dit qu'il faudrait pour cela un nombre infini d'axiomes de départ.

Cette méthode reste utilisable, mais nécessite en général que le développeur écrive lui-même la preuve, qui sera alors simplement vérifiée par l'outil de preuve automatique, ou au moins qu'il fournisse les différents axiomes dont l'outil aura besoin, et guide le processus de vérification.

Cette approche a été introduite par Hoare [Hoa69], et est utilisée dans des outils comme COQ [HKPM97, CH88] ou PVS [ORS92].

Le model-checking : il s'agit ici d'utiliser un algorithme pour vérifier directement que le modèle satisfait la spécification souhaitée, quels que soient les comportements de l'utilisateur.

Le model-checking possède sur les preuves automatiques l'avantage qu'une fois que le programme est modélisé, aucune intervention humaine n'est nécessaire. Un outil, le *model-checker*, s'occupe d'effectuer tous les calculs et de retourner une réponse au problème posé.

D'autre part, en cas de réponse négative (c'est-à-dire si le modèle ne satisfait pas la spécification), le *model-checker* retourne un contre-exemple : une exécution précise du modèle qui ne satisfait pas la spécification. Ce contre-exemple est particulièrement utile, parce qu'il permet de situer la source de l'erreur dans un modèle souvent complexe.

Le model-checking a été présenté notamment par [QS82, CES86, VW86].

1.1.3 Méthodes formelles ou tests, que choisir ?

Si les méthodes formelles permettent de garantir la correction d'un modèle, pourquoi ne les utilise-t-on pas systématiquement et exclusivement ?

En fait, celles-ci travaillent justement sur un modèle et non sur le programme lui-même. En effet elles ont besoin que le programme soit réécrit sous une forme qu'elles puissent comprendre. D'autre part, les programmes à vérifier sont le plus souvent trop gros à traiter, les méthodes formelles étant des procédés coûteux en temps et en ressources. Utiliser un modèle permet de simplifier un programme en ne gardant que la partie du code qui influe sur la spécification qui doit être vérifiée.

Ainsi, par souci d'efficacité, il est préférable de commencer par supprimer toutes les erreurs que les procédures habituelles de tests peuvent déceler (sauf dans le cas d'un système matériel, où on préférera la simulation). Une fois cette première étape réalisée, on peut utiliser les méthodes formelles, et donc modéliser le programme. La modélisation pouvant en effet être coûteuse, cette procédure permet d'éviter de recalculer un modèle à chaque fois qu'une petite erreur a été corrigée.

Ensuite, la modélisation d'une application peut rarement être effectuée de façon automatique, et à partir du moment où la main de l'homme intervient, il est possible qu'une erreur soit commise et que le modèle obtenu ne modélise justement pas le comportement du programme. Dans ce cas même si le model-checking assure que la spécification est vérifiée par le modèle, le programme peut encore être incorrect. Les tests, effectués directement sur le programme, peuvent permettre de déceler une telle erreur.

Ainsi, il apparaît que les méthodes formelles doivent être utilisées conjointement avec les méthodes classiques de test. La combinaison des deux stratégies permet de multiplier l'efficacité de la détection d'erreurs.

Nous nous restreindrons par la suite uniquement au cadre du model-checking.

1.2 Le model-checking

Les premiers algorithmes de model-checking [CE81, QS82, LP85, CES86, VW86] sont apparus dans les années 80, et ont servi à l'élaboration de multiples outils de model-checking. Avec l'amélioration des techniques utilisées, le model-checking s'est vite développé et est maintenant utilisé à large échelle, en particulier dans l'industrie pour la vérification de systèmes critiques. Les domaines d'application peuvent être aussi bien les systèmes

matériels, comme les circuits électroniques, ou les systèmes logiciels, comme les protocoles de communication par exemple [Hol91, FHS95, Bar95, JLS96, RL97, LS97, JLM+98, DT98]. Certaines sociétés (INTEL, NASA, Lucent, etc.) ne se sont pas contentées de recruter des experts en model-checking mais ont aussi développé et commercialisé leur propre model-checker.

Nous allons maintenant détailler de façon plus approfondie le processus de model-checking : comment modéliser une application, comment exprimer des spécifications, et les algorithmes de model-checking.

1.2.1 Le modèle

Comme nous l'avons déjà vu, un outil de model-checking ne travaille pas sur un programme, mais sur une modélisation de celui-ci. La modélisation d'un programme consiste à construire, dans un cadre formel précis et imposé par le model-checker, un objet symbolique dont le comportement est aussi proche que possible de celui du programme, et qui est particulièrement fidèle dès que des données ayant un lien avec la spécification sont en cause. Certaines applications permettent de générer automatiquement un modèle à partir du code source de certains programmes. Il est cependant en général utile d'effectuer une simplification du modèle obtenu, en ôtant tout ce qui n'a pas de rapport avec la spécification, afin de réduire le coût ultérieur du model-checking.

Un modèle peut aussi être construit à partir d'un simple cahier des charges qui décrit de façon informelle un algorithme. La vérification de ce modèle permettra ainsi de vérifier la validité des contraintes du cahier des charges, avant de commencer l'implémentation du programme lui-même.

Suivant la nature du programme et le model-checker utilisé, le modèle pourra être un simple système de transitions, ou aura une forme plus évoluée, permettant de modéliser la communication entre processus par exemple. Un modèle peut éventuellement être infini, si son nombre de configurations n'est pas borné.

1.2.2 La spécification

La spécification exprime, une fois encore dans un contexte formel imposé par le model-checker, une ou plusieurs propriétés que le modèle doit vérifier.

Le langage d'expression d'une spécification se doit en général d'inclure les prédicats de la logique. Cependant il est courant de devoir exprimer des propriétés incluant des contraintes temporelles. Inclure des contraintes en temps réel complique fortement le model-checking. C'est pourquoi on s'intéresse la plupart du temps simplement à l'ordre dans lequel les événements s'enchaînent : on parle alors de logique temporelle [Pnu77, Eme90]. Celle-ci permet d'exprimer des contraintes naturelles comme *«jamais il n'y aura une erreur»* ou encore *«toute demande sera suivie d'une réponse»*. Le model-checking de la logique temporelle a été abordé pour la première fois au début des années 1980 [CE81, QS82].

Plusieurs logiques temporelles existent, avec des pouvoirs d'expression différents. Deux familles de logiques temporelles sont couramment utilisées : les premières sont des logiques

temporelles linéaires, comme LTL [Pnu81, Pnu86, MW81], et les secondes sont des logiques temporelles arborescentes, comme CTL [CE81, EH82, CES86].

1.2.3 Les algorithmes de model-checking

Un model-checker est un outil automatique qui agit de la façon suivante : il prend en entrée un modèle exprimé dans un formalisme imposé, et une spécification qui exprime une propriété que doivent vérifier certaines données du modèle. Il effectue ensuite un calcul à partir de ces données, et peut produire deux résultats différents : soit *toutes* les exécutions du modèle satisfont la spécification, et le résultat est positif, soit au moins une exécution du modèle ne satisfait pas la spécification, et dans ce cas le résultat est négatif et le model-checker donne cette exécution ou une simplification de celle-ci comme un contre-exemple d'exécution non satisfaisante.

A partir de ce contre-exemple, l'utilisateur peut essayer de corriger la source du problème puis effectuer une nouvelle vérification du modèle. La source du problème peut être soit une erreur de l'application ou du cahier des charges, soit une erreur de modélisation, soit enfin une erreur de spécification. Le contre-exemple permet de retrouver ces trois types d'erreurs.

Le calcul, effectué par l'implémentation d'un algorithme de vérification, sera plus ou moins complexe suivant le format accepté pour le modèle et la spécification. La complexité se traduit par un coût en temps de calcul d'une part, et en espace mémoire utilisé d'autre part. En règle générale, plus le formalisme permet d'exprimer des propriétés riches, plus le calcul sera coûteux.

Même sur des modèles finis, la complexité des systèmes étant de plus en plus grande, la taille des modèles a fortement augmenté : c'est ce que l'on appelle l'*explosion combinatoire*. Cette augmentation exponentielle de la difficulté ne peut pas être compensée par l'augmentation de la puissance des machines, et de nombreux travaux ont donc été effectués pour trouver des solutions : ne représenter qu'une partie de modèle en mémoire [JJ89, FJJM92], réduire le nombre de chemins à explorer [God90, GW94, Pel94, KM97, CJM00], décomposer le système à vérifier [Pnu84, Win90], utiliser des techniques d'abstraction [GL93, Lon93], ou utiliser une représentation symbolique plus compacte [CBM90, BCM+90, Bry92].

Dans certains cas, en particulier quand le modèle est infini, le problème est *indécidable*. Il existe des model-checkers qui essaient de résoudre ces problèmes, mais sur certaines données ils sont incapables de retourner un résultat. Ils utilisent des méthodes comme l'interprétation abstraite [CC77, CH78, BBF+00] ou l'accélération [BW94, BGWW97, BH97]. Dans le cas des modèles finis, les logiques temporelles couramment utilisées sont en général décidables.

1.3 Contenu de la thèse

Le contenu de cette thèse se restreint à la vérification des formules de logiques temporelles linéaires sur des modèles finis. Les modèles finis ne permettent pas de modéliser

toutes les applications, mais ils permettent de rester dans un cadre décidable. Les applications citées comme critiques ci-dessus, comme les circuits électroniques ou les protocoles de communications, peuvent en général être modélisés de façon finie. D'autre part, en fonction de la spécification à vérifier, certains modèles infinis peuvent être simplifiés sans changer leur comportement vis-à-vis de la spécification, et ainsi devenir finis, en bornant la valeur de certaines variables par exemple.

Dans la première partie, les notions de base seront introduites : les logiques temporelles LTL et PLTL d'une part, et les principes généraux de model-checking sur ces deux logiques.

Dans la deuxième partie, nous nous intéresserons au model-checking de la logique LTL. Le contenu de cette partie a été publié dans [GO01].

Dans la troisième partie, nous aborderons le problème des opérateurs passés, avec le model-checking de la logique PLTL. Le contenu de cette partie a été publié dans [GO03a], une version longue est aussi disponible [GO03b].

Première partie

Notions de base

Chapitre 2

Logiques temporelles linéaires

Lors de la vérification d'un système, il peut être nécessaire d'exprimer des propriétés sur le comportement dynamique du système en vue de vérifier celui-ci. Selon les systèmes à vérifier la spécification des propriétés peut être très différente. Les systèmes temps-réel nécessitent de pouvoir spécifier des propriétés portant sur des intervalles de temps définis par des variables temporelles, les horloges. Dans des cas plus simples comme ceux que nous allons étudier, les propriétés font simplement référence à l'ordonnancement des événements dans le temps. Dans ce cas on utilise des *logiques temporelles*.

Pnueli [Pnu77] a introduit le premier les notations de la logique temporelle, et définissant des opérateurs traduisant les propriétés que l'on peut exprimer en langue naturelle. Par exemple, pour la logique temporelle linéaire (LTL), les opérateurs sont les suivants : toujours, suivant, inévitablement, jusqu'à, ... Ceci rend relativement simple la formalisation d'une propriété en logique temporelle.

La logique la plus utilisée est probablement la logique temporelle arborescente (CTL) [CE81, EH82, CES86], qui permet d'exprimer des propriétés sur des arbres d'exécution. Elle possède en effet un avantage : la vérification de propriétés exprimées en CTL se fait en temps polynômial en la taille de la formule : elle est P-complète (voir section 3.2.1). Toutefois cette logique présente des lacunes, certaines propriétés naturelles ne pouvant pas être exprimées dans cette logique [JMC94, Sch97].

LTL permet d'exprimer des propriétés sur des séquences d'états, dites *formules de chemin*, ce qui la rend plus naturelle et plus simple à comprendre que CTL. Elle possède aussi certaines lacunes, mais moins gênantes que celles de CTL. Cependant la vérification de formules LTL est beaucoup plus coûteuse [CGH94, Var01], et nécessite un temps exponentiel (elle est PSPACE-complète). La contrepartie du fait que la vérification de formules LTL est difficile, est que la recherche est plus active, dans le but de trouver des algorithmes de model-checking toujours plus efficaces.

2.1 LTL

2.1.1 Définitions

Soit AP un ensemble de propositions atomiques, et soit $\Sigma = 2^{\mathrm{AP}}$. Nous notons Σ^ω l'ensemble des mots infinis sur l'alphabet Σ.

Définition 2.1 (*syntaxe*)

Les formules LTL sont définies ainsi :
- $\bot$ (*false*), et tous les éléments p dans AP sont des formules LTL,
- si φ et ψ sont des formules LTL, alors $\neg\varphi$ (*not* φ), $\varphi \vee \psi$ (φ *or* ψ), $\mathsf{X}\,\varphi$ (*next* φ) et $\varphi \,\mathsf{U}\, \psi$ (φ *until* ψ) sont des fomules LTL.

La sémantique de LTL définit si une exécution d'un système donné satisfait une formule. En fait, la sémantique dépend uniquement des propositions atomiques qui sont vraies dans chaque état de cette exécution. C'est pourquoi nous définissons la sémantique de LTL sur Σ^ω.

Définition 2.2 (*sémantique*)

Soient $u = u_1u_2\ldots$ un mot de Σ^ω, φ une formule LTL, et i un entier tel que $i \geq 1$. La relation $u, i \models \varphi$ (φ *est vraie à la position* i), est définie comme suit :
- $u, i \not\models \bot$,
- $u, i \models p$ si $p \in u_i$,
- $u, i \models \neg\,\varphi$ si $u, i \not\models \varphi$,
- $u, i \models \varphi \vee \psi$ si $u, i \models \varphi$ ou $u, i \models \psi$,
- $u, i \models \mathsf{X}\,\varphi$ si $u, i+1 \models \varphi$,
- $u, i \models \varphi \,\mathsf{U}\, \psi$ si $\exists j \geq i$ tel que $u, j \models \psi$ et $\forall i \leq k < j$, $u, k \models \varphi$.

On note $u \models \varphi$ au lieu de $u, 1 \models \varphi$.
Le *langage* d'une formule est défini par $\mathcal{L}(\varphi) = \{u \in \Sigma^\omega \mid u \models \varphi\}$.

Nous définissons ici la taille temporelle d'une formule, qui est le critère déterminant pour la taille des automates obtenus par la suite.

Définition 2.3 (*taille temporelle d'une formule*)

La *taille temporelle* $|\varphi|$ mesure le nombre d'opérateurs temporels d'une formule φ. Elle est définie par induction comme suit :
- $|\top| = |\bot| = |p| = 1$ pour tout p dans AP,
- $|\neg\varphi| = |\varphi|$, $|\varphi \vee \psi| = |\varphi| + |\psi|$, $|\mathsf{X}\,\varphi| = 1 + |\varphi|$, $|\varphi \,\mathsf{U}\, \psi| = 1 + |\varphi| + |\psi|$.

Seuls les opérateurs de base on été définis ci-dessus. on définit aussi les opérateurs suivants :

L'opérateur de négation $\neg$ possède une fâcheuse tendance à provoquer une explosion combinatoire lorsqu'on travaille sur une formule LTL. Il est courant, à l'aide des opérateurs duaux, de repousser toutes les négations au niveau des prédicats.

$\top \stackrel{def}{=} \neg\bot$	(*true*)	$\varphi \,\mathsf{R}\, \psi \stackrel{def}{=} \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$	(φ *releases* ψ)
$\varphi \wedge \psi \stackrel{def}{=} \neg(\neg\varphi \vee \neg\psi)$	(φ *and* ψ)	$\mathsf{F}\,\varphi \stackrel{def}{=} \top \,\mathsf{U}\, \varphi$	(*eventually* φ)
$\varphi \rightarrow \psi \stackrel{def}{=} \neg\varphi \vee \psi$	(φ *implies* ψ)	$\mathsf{G}\,\varphi \stackrel{def}{=} \bot \,\mathsf{R}\, \varphi$	(*always* φ)
$\varphi \leftrightarrow \psi \stackrel{def}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$	(φ *equiv. to* ψ)		

Tab. 2.1 – Opérateurs additionnels pour LTL

Définition 2.4 (*forme normale négative*)

Une formule LTL qui n'utilise que $\top$, $\bot$, les éléments de AP et leurs négations, et les opérateurs $\vee$, $\wedge$, X, U, R est dite en *forme normale négative*. Une formule en forme normale négative qui n'est ni une conjonction ($\wedge$) ni une disjonction ($\vee$) est une *formule temporelle*.

Toute formule LTL peut être transformée en une formule en forme normale négative de même langage. Il suffit en effet de remplacer la négation d'un opérateur par son opérateur dual, en repoussant les négations sur les fils, et de supprimer les doubles négations. Ainsi $\neg(p \,\mathsf{R}\, (\neg q))$ sera réécrite en $(\neg p) \,\mathsf{U}\, q$. Notons que cette transformation ne modifie pas la taille temporelle de la formule. Par la suite, sauf précision contraire, nous supposerons que toute formule LTL est en forme normale conjonctive.

2.1.2 Simplification

Tous les algorithmes de traitement des formules LTL que nous allons présenter dans la suite de cette thèse ont une complexité liée à la taille temporelle de la formule LTL traitée, et pour générer un automate à partir d'une formule LTL cette complexité est plus qu'exponentielle. Il apparaît donc important de pouvoir réduire par des procédures de simplification la taille temporelle d'une formule avant d'effectuer des calculs.

De nombreux travaux ont été effectués sur ce sujet. Nous allons exposer ici une partie des simplifications que nous avons retenues, pour le fait qu'elles permettent de réduire strictement la taille temporelle d'une formule.

Nous présentons ici un tableau de formules de simplification, dont la majorité des éléments sont tirés de [Hol97, EH00, SB00]. On note $\varphi \leq \psi$ si $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\psi)$ et $\varphi \equiv \psi$ si $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$. La simplification consiste à remplacer le membre de gauche par le membre de droite dans chaque relation $\equiv$ appliquable. Chaque relation peut aussi être appliquée dans sa forme duale, et échangeant les opérateurs $\top, \wedge, \mathsf{U}, \mathsf{F}$ et leurs duaux respectifs $\bot, \vee, \mathsf{R}, \mathsf{G}$, et en remplaçant $\leq$ par $\geq$.

On notera toutefois que, pour deux formules LTL φ et ψ, déterminer si $\varphi \leq \psi$ est difficile. Pour cela, nous utilisons la méthode de [SB00], qui consiste à approximer la relation $\leq$ à partir des règles d'inférence suivantes, et de leur duales, déduites en suivant les mêmes règles que précédemment :

Prouver toutes les relations d'équivalence de la table 2.2, et prouver que chaque relation

$$\begin{array}{rcr}
\mathsf{X}\,\top \equiv \top & \mathsf{G}(\varphi\,\mathsf{R}\,\psi) \equiv \mathsf{G}\,\psi & \mathsf{X}\,\varphi \wedge \mathsf{X}\,\psi \equiv \mathsf{X}(\varphi \wedge \psi) \\
\varphi\,\mathsf{U}\,\bot \equiv \bot & \mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\mathsf{F}\,\varphi & \mathsf{X}\,\varphi\,\mathsf{U}\,\mathsf{X}\,\psi \equiv \mathsf{X}(\varphi\,\mathsf{U}\,\psi) \\
\varphi \wedge (\psi\,\mathsf{U}\,\varphi) \equiv \varphi & \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\mathsf{F}\,\varphi & (\varphi\,\mathsf{R}\,\psi) \wedge (\varphi\,\mathsf{R}\,\theta) \equiv \varphi\,\mathsf{R}\,(\psi \wedge \theta) \\
\varphi \wedge (\psi\,\mathsf{R}\,\varphi) \equiv \varphi & \mathsf{G}\,\mathsf{F}\,\varphi \vee \mathsf{G}\,\mathsf{F}\,\psi \equiv \mathsf{G}\,\mathsf{F}(\varphi \vee \psi) & (\varphi\,\mathsf{R}\,\psi) \vee (\theta\,\mathsf{R}\,\psi) \equiv (\varphi \vee \theta)\,\mathsf{R}\,\psi
\end{array}$$

$$\begin{array}{rccc}
\text{si } \varphi \leq \psi, & \varphi\,\mathsf{U}\,\psi \equiv \psi & \text{et} & \varphi \wedge \psi \equiv \varphi \\
\text{si } \neg\psi \leq \varphi, & \varphi\,\mathsf{U}\,\psi \equiv \mathsf{F}\,\psi & \text{et} & \varphi \vee \psi \equiv \top \\
\text{si } \varphi \leq \psi, & \varphi\,\mathsf{U}\,(\psi\,\mathsf{U}\,\theta) \equiv \psi\,\mathsf{U}\,\theta & &
\end{array}$$

Tab. 2.2 – Règles de simplification pour LTL

$$\begin{array}{ll|ll}
\textit{Initialisation} : \varphi \leq \top \text{ et } \varphi \leq \varphi & & \text{si } \varphi \leq \theta \text{ ou } \psi \leq \theta, & \varphi \wedge \psi \leq \theta \\
\text{si } \varphi \leq \theta, & \varphi \leq \psi\,\mathsf{U}\,\theta & \text{si } \varphi \leq \theta \text{ et } \psi \leq \theta, & \varphi\,\mathsf{U}\,\psi \leq \theta \\
\text{si } \varphi \leq \psi \text{ et } \varphi \leq \theta, & \varphi \leq \psi \wedge \theta & \text{si } \varphi \leq \theta \text{ et } \psi \leq \chi, & \varphi\,\mathsf{U}\,\psi \leq \theta\,\mathsf{U}\,\chi
\end{array}$$

Tab. 2.3 – Approximation de la relation $\leq$

de la table 2.3 implique l'inclusion des langages des deux éléments ne pose aucune difficulté. Pour plus de détails, nous vous renvoyons à [EH00, SB00].

2.2 PLTL : LTL avec passé

Si la plupart des logiques temporelles utilisées en model-checking pour spécifier les propriétés des systèmes sont des logiques *pur futur*, l'utilisation d'opérateurs du passé rend les spécifications plus simples et plus naturelles.

Les opérateurs du passé sont présents dans les travaux des philosophes sur la logique temporelle [Pri51, RU71], mais les informaticiens les ont supprimés dans un souci de minimalité, après les travaux de [GPSS80] qui prouvent que, dans la cas d'un futur infini et d'un passé fini, les opérateurs du passé n'ajoutent pas d'expressivité.

Plus récemment, des nouveaux arguments sont apparus plaidant la réintroduction des opérateurs de passé. Le premier argument est que si les opérateurs du passé n'ajoutent pas de pouvoir expressif, le prix de leur élimination est élevé, causant une explosion non-élémentaire sur la taille des formules considérées [LPZ85]. De fait, de nombreuses spécifications naturelles sont beaucoup plus simples à exprimer en utilisant les opérateurs du passé [KVR83]. Plus tard enfin on a prouvé que PLTL est exponentiellement plus succincte que LTL [LMS02]. C'est pourquoi nous nous intéressons ici aux formules PLTL.

La logique temporelle linéaire du passé utilise, en plus de LTL, deux nouveaux opérateurs : Y et S.

Définition 2.5 (*syntaxe*)

Les formules PLTL sont définies ainsi :
- $\bot$, et tous les éléments p dans AP sont des formules PLTL,
- si φ et ψ sont des formules PLTL, alors $\neg\varphi$, $\varphi \vee \psi$, $\mathsf{X}\,\varphi$, $\varphi\,\mathsf{U}\,\psi$, $\mathsf{Y}\,\varphi$ (*yesterday* φ) et $\varphi\,\mathsf{S}\,\psi$ (φ *since* ψ) sont des formules PLTL.

Soit $\Sigma = 2^{\mathrm{AP}}$, $i \in \mathbb{N}^*$: nous notons Σ^i l'ensemble des mots finis de longueur i sur l'alphabet Σ, Σ^+ l'ensemble des mots finis non vides, Σ^ω l'ensemble des mots infinis, et $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$. Pour $u \in \Sigma^\infty$, on note $|u|$ la taille de u ($|u| = \omega$ si $|u| \in \Sigma^\omega$).

Pour PLTL, contrairement au cas de LTL, nous traiterons toujours le cas des mots finis conjointement à celui des mots infinis. En effet, les difficultés rencontrées lors du traitement des opérateurs du passé en début de mot sont similaires à celles des opérateurs du futur en fin de mot. Ainsi si ces difficultés peuvent être évitées en se restreignant aux mots infinis pour LTL, il n'en est pas autant pour PLTL. Ainsi nous traiterons toujours, pour PLTL, le cas des mots finis.

Définition 2.6 (*sémantique*)

Soient $u = u_1u_2\ldots$ un mot de Σ^∞, φ une formule PLTL, et i un entier tel que $1 \leq i \leq |u|$. La relation $u, i \models \varphi$ (φ *est vraie à la position* i), est définie comme suit :
- $u, i \not\models \bot$,
- $u, i \models p$ si $p \in u_i$,
- $u, i \models \neg\,\varphi$ si $u, i \not\models \varphi$,
- $u, i \models \varphi \vee \psi$ si $u, i \models \varphi$ ou $u, i \models \psi$,
- $u, i \models \mathsf{X}\,\varphi$ si $i < |u|$ et $u, i+1 \models \varphi$,
- $u, i \models \mathsf{Y}\,\varphi$ si $i > 1$ et $u, i-1 \models \varphi$,
- $u, i \models \varphi\,\mathsf{U}\,\psi$ si $\exists i \leq j \leq |u|$ tel que $u, j \models \psi$ et $\forall i \leq k < j$, $u, k \models \varphi$.
- $u, i \models \varphi\,\mathsf{S}\,\psi$ si $\exists 1 \leq j \leq i$ tel que $u, j \models \psi$ et $\forall j < k \leq i$, $u, k \models \varphi$.

On note $u \models \varphi$ au lieu de $u, 1 \models \varphi$.
Le *langage* d'une formule est défini par $\mathcal{L}(\varphi) = \{u \in \Sigma^\infty \mid u \models \varphi\}$.
On définit aussi $\mathcal{L}(\varphi, i) = \{u \in \Sigma^\infty \mid u, i \models \varphi\}$.

La taille temporelle d'une formule est étendue pour les nouveaux opérateurs.

Définition 2.7 (*taille temporelle d'une formule*)

La *taille temporelle* $|\varphi|$ mesure le nombre d'opérateurs temporels d'une formule φ.
Elle est définie par induction comme suit :
- $|\top| = |\bot| = |p| = 1$ pour tout p dans AP,
- $|\neg\varphi| = |\varphi|$, $|\varphi \vee \psi| = |\varphi| + |\psi|$, $|\mathsf{X}\,\varphi| = |\mathsf{Y}\,\varphi| = 1 + |\varphi|$, $|\varphi\,\mathsf{U}\,\psi| = |\varphi\,\mathsf{S}\,\psi| = 1 + |\varphi| + |\psi|$.

Ici encore, on ajoute les opérateurs duaux $\top$, $\wedge$, $\widetilde{\mathsf{X}}$, $\widetilde{\mathsf{Y}}$, $\widetilde{\mathsf{U}}$, $\widetilde{\mathsf{S}}$. Ainsi, par exemple, $\widetilde{\mathsf{X}}\,\varphi \overset{def}{=} \neg\,\mathsf{X}\,\neg\,\varphi$ et $\varphi\,\widetilde{\mathsf{S}}\,\psi \overset{def}{=} \neg(\neg\,\varphi\,\mathsf{S}\,\neg\,\psi)$. Notons que, dans le cadre des mots finis, $\widetilde{\mathsf{X}}\,\varphi$ n'est plus equivalent à $\mathsf{X}\,\varphi$: au contraire, on a $\widetilde{\mathsf{X}}\,\varphi \equiv \neg\,\mathsf{X}\,\top \vee \mathsf{X}\,\varphi$, la formule $\neg\,\mathsf{X}\,\top$ étant

vraie, par définition, uniquement à la fin d'un mot fini. L'opérateur $\widetilde{\mathsf{U}}$ est simplement une nouvelle notation pour l'opérateur R défini pour LTL, adoptée pour des raisons d'homogénéité. On définit enfin $\mathsf{O}\,\varphi \stackrel{def}{=} \top\,\mathsf{S}\,\varphi$ (*once* φ), $\mathsf{H}\,\varphi \stackrel{def}{=} \bot\,\widetilde{\mathsf{S}}\,\varphi$ (*historically* φ : φ est toujours vraie dans le passé).

On étend alors la notion de forme normale négative pour PLTL, sous la forme que nous attendions.

Définition 2.8 (*forme normale négative*)

Une formule PLTL qui n'utilise que $\top$, $\bot$, les éléments de AP et leurs négations, et les opérateurs $\vee$, $\wedge$, X, $\widetilde{\mathsf{X}}$, Y, $\widetilde{\mathsf{Y}}$, U, $\widetilde{\mathsf{U}}$, S, $\widetilde{\mathsf{S}}$ est dite en *forme normale négative*. Une formule en forme normale négative qui n'est ni une conjonction ($\wedge$) ni une disjonction ($\vee$) est une *formule temporelle*.

2.3 Exemples

Nous présentons dans cette section plusieurs exemples de formules LTL et PLTL, choisis pour leur représentativité dans le cadre du model-checking.

$$\mathsf{G}\,\mathsf{F}\,p$$

Toujours un jour p, c'est à dire que la proposition p est vraie infiniment souvent dans le cadre des mots infinis.

$$\mathsf{G}(r \to \mathsf{F}\,g)$$

Toujours, une requête (r) est suivie ultérieurement d'une réponse (g : *grant*).

$$(\mathsf{G}\,\mathsf{F}\,p_1 \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\,p_n) \to \mathsf{G}(r \to \mathsf{F}\,g)$$

C'est une combinaison des équations précédentes : si les événements p_1, p_2, ..., p_n sont tous vrais infinement souvent, alors on exige que toute requête soit suivie d'une réponse. Les p_i pourraient représenter ici des conditions nécessaires pour que le système soit en état de fonctionnement correct et puisse garantir une réponse à toute requête.

$$\mathsf{G}(g \to \mathsf{Y}(\neg g\,\mathsf{S}\,r))$$

Toujours, au plus une réponse suit chaque requête (quand une réponse est effectuée, on vérifie qu'aucune autre réponse n'a été effectuée depuis la dernière requête). Cette spécification est beaucoup plus difficile à exprimer en LTL.

$$\mathsf{G}(p_1 \to \mathsf{O}(p_2 \wedge \mathsf{O}(p_3 \wedge \ldots \mathsf{O}(p_n) \ldots)$$

Toujours, l'événement p_1 doit être précédé de l'événement p_2, lui-même devant être précédé de p_3, et ainsi de suite, p_{n-1} devant être précédé de p_n.

Chapitre 3

Model-checking pour LTL et PLTL

Dans ce chapitre seront présentées les méthodes usuelles pour réaliser le model-checking de spécifications exprimées en logiques temporelles linéraires.

3.1 Formalismes utilisés

Dans le contexte que nous avons choisi, un modèle fini est un système de transitions. Nous le représentons par une structure de Kripke, avec un ensemble fini d'états, des transitions entre les états, et un étiquetage qui associe à chaque état l'ensemble des propositions atomiques de AP qui sont vraies dans cet état. Ainsi, une exécution dans une structure de Kripke représente une exécution du système qu'elle modélise.

Définition 3.1 (*structure de Kripke*)

Une *structure de Kripke* sur AP est un quadruplet $\mathcal{K} = (S, \rightarrow, S_0, \lambda)$ où :
- S est l'ensemble fini des *états*,
- $\rightarrow \subseteq S \times S$ est la *relation de transition*, elle est *totale* : $\forall s \in S$, $\exists s' \in S$, $s \rightarrow s'$,
- $S_0 \subseteq S$ est l'ensemble des *états initiaux*,
- $\lambda : S \rightarrow 2^{\mathrm{AP}}$ est l'*étiquetage* des états.

Une *exécution* ρ sur $\mathcal{K}$ est une suite infinie d'états $q_0, q_1 \ldots$ telle que $q_0 \in S_0$ et $\forall i \geq 0$, $q_i \rightarrow q_{i+1}$. Le mot $\lambda(q_0)\lambda(q_1)\ldots \in (2^{\mathrm{AP}})^\omega$ est l'*étiquette* de l'exécution ρ.
Le *langage* $\mathcal{L}(\mathcal{K})$ d'une structure de Kripke est l'ensemble des étiquettes des exécutions.

Un exemple de structure de Kripke est proposé sur la figure 3.1.

Nous aurons aussi besoin de la notion suivante d'automates de Büchi sur les mots infinis :

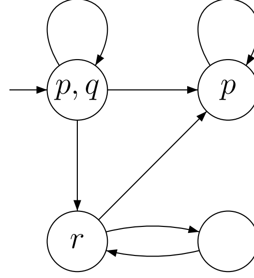


FIG. 3.1 – Exemple de structure de Kripke

Définition 3.2 (*automate de Büchi*)

Un *automate de Büchi* est un quintuplet $\mathcal{B} = (Q, \Sigma, \delta, I, R)$ où :
- Q est l'ensemble fini des *états*,
- Σ est l'*alphabet*,
- $\delta : Q \times \Sigma \to 2^Q$ est la *fonction de transition*,
- $I \subseteq Q$ est l'ensemble des *états initiaux*,
- $R \subseteq Q$ est l'ensemble des *états répétés*.

Soit $u = u_1 u_2 \ldots \in \Sigma^\omega$. Une *exécution* ρ de $\mathcal{B}$ sur u est une suite infinie $q_0, q_1, \ldots$ d'éléments de Q telle que :
- le premier état est initial : $q_0 \in I$,
- on passe d'un état au suivant en suivant la fonction de transition : $\forall i \geq 1,\ q_i \in \delta(q_{i-1}, u_i)$.

Une exécution ρ est *acceptante* si, de plus, une infinité de q_i sont dans l'ensemble R. Le *langage* $\mathcal{L}(\mathcal{B})$ d'un automate de Büchi $\mathcal{B}$ est l'ensemble des mots de Σ^ω sur lesquels il existe une exécution acceptante de $\mathcal{B}$.

3.2 Algorithme général de model-checking

Il s'agit ici de définir un algorithme de model-checking qui, prenant en entrée un formule de logique temporelle linéaire (LTL ou PLTL) φ et un modèle sous la forme d'un structure de Kripke $\mathcal{K}$, renvoie après calcul un résultat binaire : *oui* si $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$, *non* sinon. Si la réponse est *oui*, on dit que le modèle $\mathcal{K}$ satisfait la spécification φ, et on note $\mathcal{K} \models \varphi$.

3.2.1 Complexité

On connaît pour CTL un algorithme bilinéaire de model-checking c'est-à-dire de complexité $\mathcal{O}(|\mathcal{K}| \times |\varphi|)$ [CES86]. De plus, cet algorithme est relativement simple, et a été implémenté dans l'outil SMV [BCM+90, McM93].

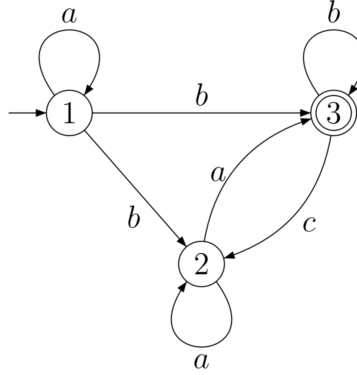


FIG. 3.2 – Exemple d'automate de Büchi

Pour LTL, on sait que le problème est décidable [Pnu77], et qu'il est PSPACE-complet [Sis83, SC85]. Ce résultat tendrait à privilégier l'utilisation de CTL pour le model-checking, mais on connaît un algorithme de complexité $\mathcal{O}(|\mathcal{K}| \times 2^{\mathcal{O}(|\varphi|)})$ [LP85], ce qui induit que dans le cas de petites formules, le facteur majeur de cette complexité est une linéarité en la taille du modèle, comme pour CTL.

3.2.2 Principe général

Il existe plusieurs méthodes de model-checking, mais la plus courante consiste à construire un automate à partir d'une formule de logique temporelle [VW86, BVW94].

Soient AP un ensemble de propositions atomiques, $\Sigma = 2^{\mathrm{AP}}$, φ une formule de logique temporelle linéaire (LTL ou PLTL) sur AP, et $\mathcal{K} = (Q, \to, I, \lambda)$ une structure de Kripke sur AP.

$$\mathcal{K} \models \varphi \iff \mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi) \iff \mathcal{L}(\mathcal{K}) \cap \overline{\mathcal{L}(\varphi)} = \emptyset \iff \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\neg\varphi) = \emptyset$$

Voici le principe de l'algorithme de model-checking engendré par cette équivalence :

- Commençons par construire à partir de la formule φ, un automate de Büchi $\mathcal{B}_{\neg\varphi}$ tel que $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$.
- Construisons ensuite, à partir de $\mathcal{K}$ et de l'automate $\mathcal{B}_{\neg\varphi}$, un automate de Büchi $\mathcal{B}'$ tel que $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi})$.
- Enfin testons si le langage de cet automate $\mathcal{B}'$ est vide. Deux cas de figure peuvent se produire :
  - si $\mathcal{L}(\mathcal{B}') = \emptyset$, alors $\mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\neg\varphi) = \emptyset$, donc $\mathcal{K} \models \varphi$,
  - si au contraire il existe un mot $u \in \mathcal{L}(\mathcal{B}')$, alors ce mot est l'étiquette d'une exécution ρ de $\mathcal{K}$ et ne satisfait pas la formule φ. Dans ce cas, $\mathcal{K} \not\models \varphi$, et ρ donne un contre-exemple, c'est-à-dire une exécution du modèle qui ne satisfait pas la spécification.

3.2.3 Explications

Nous allons maintenant revenir sur les différents points de cet algorithme : construction de $\mathcal{B}_{\neg\varphi}$, de $\mathcal{B}'$, test du vide, et enfin complexité de l'algorithme :

Construction de $\mathcal{B}_{\neg\varphi}$ Cette construction est le point essentiel de l'algorithme. C'est à celle-ci que se consacre le reste de cette thèse, pour des formules de LTL et de PLTL, et nous aurons donc l'occasion d'y revenir à plusieurs reprises.

Construction de $\mathcal{B}'$ Il s'agit de construire un automate de Büchi reconnaissant l'intersection du langage d'une structure de Kripke et du langage d'un automate de Büchi. Cette construction est en fait une application du *produit synchronisé* de deux automates de Büchi au cas plus simple où l'un des deux est une structure de Kripke :

Définition 3.3 (*produit synchronisé*)

Soient $\mathcal{K} = (S, \to, S_0, \lambda)$ une structure de Kripke sur AP et $\mathcal{B} = (Q, \Sigma, \delta, I, R)$ un automate de Büchi tel que $\Sigma = 2^{\mathrm{AP}}$. Nous définissons le *produit synchronisé* de $\mathcal{K}$ et de $\mathcal{B}$, comme l'automate de Büchi défini par $\mathcal{K} \times \mathcal{B} = (Q', \Sigma, \delta', I', R')$ avec :
- $Q' = (\{\rhd\} \cup S) \times Q$,
- $\forall s \in S$, $\forall q \in Q$, $\forall a \in \Sigma$,
  $\delta'((\rhd, q), a) = \{(s', q') \in S \times Q \mid s' \in S_0,\ \lambda(s') = a,\ q' \in \delta(q, a)\}$,
  $\delta'((s, q), a) = \{(s', q') \in S \times Q \mid s \to s',\ \lambda(s') = a,\ q' \in \delta(q, a)\}$,
- $I' = \{\rhd\} \times I$,
- $R = S \times R$.

Le produit synchronisé correspond à ce que nous attendions : il reconnaît l'intersection des deux langages.

Proposition 3.1

$\mathcal{L}(\mathcal{K} \times \mathcal{B}) = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B})$.

Démonstration. Ce résultat est un conséquence du classique produit synchronisé entre deux automates de Büchi. Voir [CGP99] par exemple. □

Test du vide Le *test du vide* consiste à tester si le langage d'un automate de Büchi est vide. Pour ça, il existe plusieurs méthodes. Dans la première (voir [LP85]), on détermine avec l'algorithme de Tarjan [Tar72] les *composantes fortement connexes* accessibles de l'automate. Le langage de l'automate est non vide si et seulement si l'une de celles-ci contient un état répété. L'autre méthode (voir [CVWY91, HPY96]) est une *double recherche en profondeur* dans le graphe de l'automate, qui permet de chercher l'ensemble des états accessibles qui appartiennent à une boucle. Le langage de l'automate est non vide si et seulement si un de ces états est répété.

Dans le cas où le langage de l'automate est non vide, on peut trouver un mot infini accepté par l'automate, sous la forme $u.v^\omega$ où u et v sont deux mots de Σ^+, ce qui

permet de décrire ce mot infini de façon finie. Dans le deuxième algorithme, ce mot est donné directement par la pile d'appels récursifs : le mot u correspond au chemin d'accès d'un état initial à l'état répété, et le mot v à la boucle sur l'état répété.

Complexité de l'algorithme Nous admettons pour l'instant que pour une formule LTL φ, la première étape permet en temps $2^{\mathcal{O}(|\varphi|)}$ d'obtenir un automate $\mathcal{B}_{\neg\varphi}$ de taille $2^{\mathcal{O}(|\varphi|)}$. L'automate produit $\mathcal{B}'$ n'a pas réellement besoin d'être construit, on connaît ses états et sa fonction de transition. Le test du vide se fait en temps $\mathcal{O}(|\mathcal{B}'|) = \mathcal{O}(|\mathcal{K}| \times |\mathcal{B}_{\neg\varphi}|) = \mathcal{O}(|\mathcal{K}| \times 2^{\mathcal{O}(|\varphi|)})$. On retrouve bien la complexité annoncée précédemment.

3.3 De la logique temporelle aux automates

Un des points centraux de l'algorithme général de model-checking que nous venons de présenter est la transformation d'une formule de logique temporelle en un automate qui reconnaît son langage. Il en résulte un intérêt marqué de la recherche dans ce domaine.

Le point de départ est assez clairement le travail de Büchi sur la décidabilité des théories du premier et du second ordre monadique d'un successeur [Büc62]. Ces résultats de décidabilité sont obtenus à travers une transformation en automates sur les mots infinis. Cette transformation est non-élémentaire, mais c'est le mieux que l'on puisse faire. La logique temporelle linéaire peut être traduite en théorie de premier ordre d'un successeur, et donc en automates de Büchi.

Ce résultat existentiel a été longtemps considéré comme suffisant, mais avec l'usage croissant des logiques temporelles en informatique, on s'est enfin rendu compte que cette transformation non-élémentaire pouvait et devait être améliorée pour LTL. Et de fait, une simple exponentielle suffit : la construction proposée s'appelle la méthode des tableaux [WVS83, VW94].

3.3.1 La méthode des tableaux

Cet algorithme, la *méthode des tableaux*, a été implémenté la première fois par Fujita en 1985 dans [FTM85]. Il applique les résultats théoriques présentés dans [WVS83, LP85], et transforme ainsi une formule LTL en un automate de Büchi de même langage. Ce qui est désigné par *tableau* est l'automate ainsi généré (même si les auteurs ne parlent pas d'automate).

C'est le premier algorithme pratique qui a été proposé pour effectuer cette transformation. Il est issu d'un résultat théorique et fait de nombreux calculs inutiles. De nombreuses variantes plus efficaces ont été proposées depuis, mais la compréhension de l'algorithme d'origine est probablement plus simple, et c'est pour cela qu'il est présenté ici. Il a cependant légèrement été adapté afin de pouvoir correspondre aux hypothèses effectuées dans la section précédente. Pour plus de détails, voir [Wol01] par exemple.

On suppose ici, exceptionnellement, que les formules LTL sont écrites uniquement à l'aide des opérateurs $\neg$, $\vee$, X, et U et des éléments de AP. Elles ne sont donc pas en forme

normale négative.

Nous définissons dans un premier temps la clôture d'un formule, qui est l'ensemble des formules que nous serons amenés à manipuler dans la méthode des tableaux.

Définition 3.4 (*clôture*)

La *clôture* $\mathrm{CL}(\varphi)$ d'une formule LTL φ est le plus petit ensemble contenant toutes les sous-formules de φ, $\mathsf{X}(\psi_1 \mathsf{U} \psi_2)$ pour chaque sous-formule $\psi_1 \mathsf{U} \psi_2$, et clos par l'opérateur '$\neg$' (on identifie $\neg\neg\psi$ avec ψ).

Nous définissons ensuite la notion d'atome : un atome est un sous-ensemble "cohérent" de la clôture d'une formule. Par exemple il ne contient pas à la fois une formule et sa négation, mais contient toujours l'une des deux.

Définition 3.5 (*atome*)

Un *atome* α de φ est un sous-ensemble de $\mathrm{CL}(\varphi)$ qui respecte les règles suivantes, dites règles de *cohérence* :

- $\forall \psi \in \mathrm{CL}(\varphi)$, $\psi \in \alpha \iff \neg\psi \notin \alpha$,
- $\forall \psi_1 \vee \psi_2 \in \mathrm{CL}(\varphi)$, $\psi_1 \vee \psi_2 \in \alpha \iff \psi_1 \in \alpha$ ou $\psi_2 \in \alpha$,
- $\forall \psi_1 \mathsf{U} \psi_2 \in \mathrm{CL}(\varphi)$, $\psi_1 \mathsf{U} \psi_2 \in \alpha \iff \psi_2 \in \alpha$ ou ($\psi_1 \in \alpha$ et $\mathsf{X}(\psi_1 \mathsf{U} \psi_2) \in \alpha$).

$\mathrm{Atom}(\varphi)$ est l'ensemble des atomes de φ.

Nous allons maintenant définir l'automate de Büchi $\mathcal{B}_\varphi$. L'automate aura pour états les atomes de φ, et voici l'intuition qui se cache derrière sa construction : supposons que l'automate $\mathcal{B}_\varphi$ lise un mot $u_1 u_2 \ldots$ sur l'exécution $\alpha_1, \alpha_2, \ldots$ A chaque étape i, le mot $u_i u_{i+1} \ldots$ satisfait toutes les formules de l'état α_i (et aucune formule de $\mathrm{CL}(\varphi) \backslash \alpha_i$, par définition). Les formules futures du type $\mathsf{X}\,\psi$ sont donc devinées, et vérifiées à l'étape suivante par la fonction de transition.

Définition 3.6 ($\varphi \to \mathcal{B}_\varphi$)

Soit φ une formule LTL sur AP. L'automate $\mathcal{B}_\varphi = (Q, \Sigma, \delta, I, \mathcal{R})$ est défini par :

- $Q = \mathrm{Atom}(\varphi)$,
- $\Sigma = 2^{\mathrm{AP}}$,
- $\delta(\alpha, \alpha \cap \mathrm{AP}) = \{\alpha' \in \mathrm{Atom}(\varphi) \mid \forall \mathsf{X}\,\psi \in \mathrm{CL}(\varphi), \mathsf{X}\,\psi \in \alpha \iff \psi \in \alpha'\}$, et $\forall a \neq (\alpha \cap \mathrm{AP})$, $\delta(\alpha, a) = \emptyset$,
- $I = \{\alpha \in \mathrm{Atom}(\varphi) \mid \varphi \in \alpha\}$
- $\mathcal{R} = \{R_{\psi_1 \mathsf{U} \psi_2} \mid \psi_1 \mathsf{U} \psi_2 \in \mathrm{CL}(\varphi)\}$ où $R_{\psi_1 \mathsf{U} \psi_2} = \{\alpha \in \mathrm{Atom}(\varphi) \mid \psi_2 \in \alpha$ ou $\psi_1 \mathsf{U} \psi_2 \notin \alpha\}$.

Remarque.

Cet automate généralise les automates de Büchi : une exécution acceptante doit comporter un nombre infini d'états dans chaque élément de $\mathcal{R}$.

Pour gérer cette généralisation, on peut soit convertir cet automate en un automate de Büchi classique (selon une méthode similaire à celle présentée dans la section 6.1), soit modifier l'algorithme de model-checking de la façon suivante : le produit synchronisé est calculé de la même façon, mais au moment du test du vide, il faut qu'une des composantes fortement connexes contienne un état dans *chacun* des éléments de $\mathcal{R}$. En ce qui concerne la double recherche en profondeur, lorsqu'une boucle sur un état est trouvée, il faut vérifier que chacun des éléments de $\mathcal{R}$ est visité par au moins un des états de la boucle.

3.3.2 Méthodes incrémentales

La méthode citée précédemment, dite méthode des tableaux *déclaratifs*, possède l'inconvénient majeur de construire *tous* les atomes de la spécification comme états. Si cette solution est effectivement de complexité globale optimale (PSPACE), elle est en fait exponentielle dans le pire comme dans le meilleur des cas, il apparaît assez clairement qu'elle peut être fortement améliorée.

La méthode des tableaux incrémentaux cherche à pallier au problème du trop grand nombre d'états de l'automate généré, en ne générant que des états accessibles, les ajoutant à l'automate au fur et à mesure. La méthode est plus complexe à comprendre et à analyser, mais les résultats sont bien meilleurs du point de vue de l'implémentation.

Cette méthode, dont l'idée prend peu à peu forme dans [MW84, KMMP93], est améliorée dans [GPVW95]. Cette dernière version est la plus connue, puisque c'est celle qui a été utilisée dans le logiciel SPIN [Hol97]. Elle a par la suite été améliorée à maintes reprises (voir [DGV99, EH00, SB00] par exemple).

Pour reprendre grossièrement le principe de cette méthode, les états sont les mêmes que dans la méthode des tableaux déclaratifs, mais on commence uniquement avec un seul état dans lequel φ doit être vraie, et ensuite quand un état a deux possibilités pour satisfaire une formule, il est dupliqué. Par exemple, un état devant satisfaire $\psi_1 \mathsf{U} \psi_2$ sera séparé en un état devant satisfaire ψ_2, et un autre devant satisfaire ψ_1 et $\mathsf{X}(\psi_1 \mathsf{U} \psi_2)$.

Deuxième partie

Model-checking efficace pour LTL

Introduction

Motivations

Dans le domaine du model-checking LTL, SPIN [Hol97] est l'outil le plus populaire. Cependant, l'algorithme qu'il utilise, la méthode des tableaux incrémentaux (voir section 3.3.2, ou [GPVW95]), se trouve être particulièrement lent et gourmand en ressources mémoire dans certains cas. Et c'est aussi le cas pour des formules LTL qui sont couramment utilisées en model-checking.

En particulier, prenons le cas des conditions d'équité : une condition d'équité est une formule qui demande qu'un “bon” état soit visité infiniment souvent au cours d'une exécution acceptante. Il est courant qu'une spécification LTL comporte un certain nombre de conditions d'équité. Or dès que le nombre de conditions d'équité dépasse 5, SPIN se trouve dans l'incapacité de générer un automate de Büchi reconnaissant le langage de la formule, à la fois à cause du temps de calcul nécessaire et de l'espace mémoire nécessaire à ce calcul.

L'algorithme de SPIN a été amélioré dans plusieurs nouvelles implémentations plus récentes, comme LTL2AUT [DGV99], EQLTL [EH00], ou WRING [SB00]. Ces nouvelles versions n'ont pas modifié le cœur de l'algorithme, mais ont préféré améliorer d'une part la simplification de la spécification avant son traitement, et d'autre part la simplification de l'automate de Büchi généré. Ces modifications sont effectivement très efficaces, mais la partie centrale de l'algorithme, qui transforme la formule LTL simplifiée en automate de Büchi, continue à éprouver des difficultés sur certains types de formules usuelles en model-checking.

Approche utilisée

Notre approche, suite à ces constatations, a été la suivante : chercher une nouvelle méthode pour générer un automate de Büchi à partir d'une formule LTL, qui soit différente de la méthode des tableaux. Pour cela, nous nous sommes tournés vers un autre procédé classique, qui utilise les *automates alternants* (voir [Var96] par exemple).

Ainsi, à partir d'une formule LTL φ, nous commençons par générer un automate avec au maximum $|\varphi|$ états. Le problème, est que la méthode usuellement utilisée pour générer à partir d'un automate alternant un automate de Büchi de même langage, génère pour un automate alternant à n états un automate de Büchi à $2^n \times 2^n$ états.

Pour réduire cette taille, nous exploitons la propriété suivante : l'automate alternant généré à partir d'une formule LTL est *très faible*, une propriété introduite par Rohde dans sa thèse [Roh97]. En utilisant cette particularité dont nous rappellerons le principe plus loin, nous sommes capables à partir d'un automate alternant très faible avec n états de générer un automate de Büchi généralisé avec 2^n états seulement, reconnaissant le même langage. Ce type d'automate, que nous définirons précisément en temps utile, a la particularité d'avoir des conditions d'acceptation sur les transitions et non sur les états, et d'avoir en fait plusieurs conditions d'acceptation (n au maximum dans notre cas).

Pour effectuer ensuite le model-checking, deux solutions sont possibles, tout comme

dans la remarque de la section 3.3.1 : utiliser cet automate tel quel, en en faisant le produit synchronisé avec le modèle, et en prenant en compte le nouveau type de conditions d'acceptation, ou bien transformer cet automate en un automate de Büchi et effectuer un model-checking plus classique. A cette intention nous proposons une construction qui permet d'obtenir selon un procédé classique un automate de Büchi avec au maximum $n \times (r+1)$ états acceptant le même langage qu'un automate de Büchi généralisé avec n états et r conditions d'acceptation sur les transitions.

Simplifications

La deuxième amélioration principale que nous avons proposée est la suivante : l'utilisation répétée des simplifications, et la simplification à la volée.

D'une part, notre construction se déroulant en plusieurs étapes, chaque automate est simplifié avant le passage à l'étape suivante. Ceci réduit de façon importante le nombre d'états et de transitions, et se répercute sur la taille de l'automate obtenu à l'étape suivante. Ceci est particulièrement sensible pour les automates de Büchi généralisés.

D'autre part, nous utilisons un procédé de simplification *à la volée*, c'est-à-dire que tous nos automates sont simplifiés au fur et à mesure de leur construction. Cette méthode est une grande amélioration du système usuel de simplification *a posteriori*, puisque à la fois l'espace mémoire occupé par l'automate, et le temps nécessaire à sa construction sont très fortement réduits, comme nous le verrons plus loin.

Implémentation

A partir de cet algorithme, nous avons implémenté un outil LTL2BA, dont nous avons pu comparer les performances avec les autres outils proposés pour le même usage. Notre outil s'est avéré bien plus efficace que tous ses concurrents, pour le temps de calcul et surtout pour l'espace mémoire nécessaire, tout en donnant la plupart du temps en résultat un automate de taille inférieure à celle de l'automate obtenu avec SPIN. Nous avons comparé les différents outils à la fois sur des formules usuelles pour le model-checking, et sur des formules générées aléatoirement.

Plan de la partie

Cette partie s'organise comme suit :

Dans le chapitre 4, nous définirons les automates alternants et leurs propriétés, ainsi que la méthode pour obtenir un automate alternant à partir d'une formule LTL.

Dans le chapitre 5, nous définirons notre notion d'automates de Büchi généralisés, et comment ils sont obtenus à partir des automates alternants très faibles.

Dans le chapitre 6, nous exposerons deux approches différentes pour effectuer le model-checking d'une formule LTL à partir de l'automate de Büchi généralisé obtenu.

Dans le chapitre 7, nous parlerons plus en détail de notre implémentation LTL2BA, avec nos optimisations de structures de données, nos méthodes de simplification, et enfin les résultats expérimentaux obtenus.

L'ensemble des sujets abordés dans cette partie a fait l'objet de la rédaction et de la publication d'un article [GO01], et d'un exposé à la conférence CAV'01.

Chapitre 4

Automates alternants

4.1 Définitions

Dans cette section nous définissons les automates alternants, ainsi que les automates alternants faibles et très faibles. Pour ces définitions nous introduisons les notions de combinaisons booléennes positives, et de Q-forêts.

4.1.1 Combinaisons booléennes positives

Définition 4.1 (*combinaisons booléennes positives*)

Soit Q un ensemble fini. L'ensemble des *combinaisons booléennes positives* sur Q, noté $\mathcal{B}^+(Q)$, et le plus petit ensemble contenant les éléments de Q, $\top$, $\bot$, et clos par les opérateurs $\wedge$ et $\vee$.
Soient $Q' \subseteq Q$ et $J \in \mathcal{B}^+(Q)$. La relation $Q' \models J$ (Q' *satisfait* J) est définie comme suit :

- $Q' \models \top$ et $Q' \not\models \bot$,
- $\forall J \in Q$, $Q' \models J$ ssi $J \in Q'$,
- $Q' \models J_1 \wedge J_2$ ssi $Q' \models J_1$ et $Q' \models J_2$
- $Q' \models J_1 \vee J_2$ ssi $Q' \models J_1$ ou $Q' \models J_2$

Les deux propriétés suivantes sont vérifiées par la relation de satisfaction. Elles seront utilisées ultérieurement :

Proposition 4.1

Soient $Q'_1, Q'_2 \subseteq Q$ et $J_1, J_2 \in \mathcal{B}^+(Q)$.

1. *Si $Q'_1 \subseteq Q'_2$ et $Q'_1 \models J_1$, alors $Q'_2 \models J_1$.*
2. *Si $Q'_1 \models J_1$ et $Q'_2 \models J_2$, alors $Q'_1 \cup Q'_2 \models J_1 \wedge J_2$.*

Démonstration.

La preuve du (1) est très simple et se fait par induction sur J_1.

À partir du résultat du (1), on sait que, si $Q'_1 \models J_1$ et $Q'_2 \models J_2$, alors $Q'_1 \cup Q'_2 \models J_1$ et $Q'_1 \cup Q'_2 \models J_2$. La définition de la relation de satisfaction permet de conclure pour (2). □

4.1.2 Automates alternants

Les automates alternants ont été définis dans [CKS81, MS84, MS87, MS95]. S'ils ont la même expressivité que les automates de Büchi, ils sont exponentiellement plus succincts que ceux-ci : transformer un automate alternant en un automate de Büchi implique nécessairement une explosion exponentielle du nombre d'états [DH94].

Définition 4.2 (*automate alternant*)

Un *automate alternant* est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ est la *fonction de transition*,
- $I \in \mathcal{B}^+(Q)$ est la *condition initiale*,
- $R \subseteq Q$ est l'ensemble des *états répétés*.

Ensuite, Muller et al. ont introduit la notion d'automates alternants faibles [MSS86, KV97], suivis par Rohde, qui définit dans sa thèse les automates alternants très faibles, pour les utiliser dans ses travaux sur les mots transfinis [Roh97].

Définition 4.3 (*automate alternant [très] faible*)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant.
$\mathcal{A}$ est dit *faible* s'il existe un partition $Q_1, \ldots, Q_n$ de Q et un ordre partiel $\preccurlyeq$ sur cette partition tels que :
- $\forall i, j$, $\forall (q, q') \in Q_i \times Q_j$, si $\exists a \in \Sigma$ tel que q' apparaît dans $\delta(q, a)$, alors $j \preccurlyeq i$,
- $\forall i$, $Q_i \subseteq R$ ou $Q_i \cap R = \emptyset$.

$\mathcal{A}$ est dit *très faible* si, de plus, tous les éléments de la partition ont pour cardinal 1. Autrement dit, il existe un ordre partiel $\preccurlyeq$ sur Q tel que $\forall q, q' \in Q$, si $\exists a \in \Sigma$ tel que q' apparaît dans $\delta(q, a)$, alors $q' \preccurlyeq q$.

4.1.3 Q-forêt, exécution, langage

Pour définir les exécutions d'un automate alternant, nous avons besoin de la notion de Q-forêt. En effet, une exécution d'un automate alternant aura une structure arborescente et non linéaire comme c'est le cas pour les automates de Büchi rencontrés précédemment.

Définition 4.4 (*Q-forêt*)

Une Q-forêt est une forêt étiquetée (V, E, σ) (une forêt est un ensemble d'arbres disjoints) telle que :

- V est l'ensemble des *nœuds* (vertices),
- $E \subseteq V \times V$ est l'ensemble des *arêtes* (edges),
- $\sigma : V \to Q$ est l'*étiquette* d'un nœud,

On utilisera les notations suivantes :

- Γ est l'ensemble des *racines* de la forêt :
  $\Gamma = \{y \in V \mid \forall x \in V,\ (x, y) \notin E\}$
- $\nu : V \to \mathbb{N}^*$ est la *profondeur* d'un nœud :
  $\forall x \in \Gamma,\ \nu(x) = 1$ et $\forall (x, y) \in E,\ \nu(y) = \nu(x) + 1$,
- $E(x) = \{y \in V \mid (x, y) \in E\}$,
- $E^+(x) = \{y \in V \mid (x, y) \in E^+\}$, E^+ étant la clôture transitive de E,
- $E^*(x) = \{x\} \cup E^+(x)$,
- $\sigma(X) = \{\sigma(x) \mid x \in X\}$ pour $X \subseteq V$.

Nous pouvons maintenant définir les exécutions d'un automate alternant, et le langage qu'il reconnaît.

Définition 4.5 (*exécution, langage d'un automate alternant*)

Soient $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant et $u = u_1 u_2 \ldots \in \Sigma^\omega$.
Une *exécution* de l'automate $\mathcal{A}$ sur le mot u est une Q-forêt $\rho = (V, E, \sigma)$ telle que :

- Les racines satisfont la condition initiale : $\sigma(\Gamma) \models I$,
- Les fils d'un nœud satisfont la fonction de transition :
  $\forall x \in V,\ \sigma(E(x)) \models \delta(\sigma(x), u_{\nu(x)})$.

Une exécution ρ est *acceptante* si, de plus, chaque branche infinie de la Q-forêt ρ possède un nombre infini de nœuds étiquetés dans R.
Le *langage* $\mathcal{L}(\mathcal{A})$ d'un automate alternant $\mathcal{A}$ est l'ensemble des mots de Σ^ω sur lesquels il existe une exécution acceptante de $\mathcal{A}$.
Par extension, pour $J \in \mathcal{B}^+(Q)$, on note $\mathcal{L}(\mathcal{A}, J)$ le langage de l'automate $\mathcal{A}$ dans lequel la condition initiale I est remplacée par J (noté $\mathcal{A}[I = J]$).

Le langage d'un automate alternant vérifie les propriétés suivantes :

Lemme 4.2

Soient $\mathcal{A}$ un automate alternant, $J_1, J_2 \in \mathcal{B}^+(Q)$:

1. $\mathcal{L}(\mathcal{A}, J_1 \vee J_2) = \mathcal{L}(\mathcal{A}, J_1) \cup \mathcal{L}(\mathcal{A}, J_2)$,
2. $\mathcal{L}(\mathcal{A}, J_1 \wedge J_2) = \mathcal{L}(\mathcal{A}, J_1) \cap \mathcal{L}(\mathcal{A}, J_2)$,

Démonstration. Soit $u \in \Sigma^\omega$:

1. $u \in \mathcal{L}(\mathcal{A}, J_1 \vee J_2)$
   ssi il existe une exécution acceptante ρ de $\mathcal{A}[I = J_1 \vee J_2]$ sur u
   ssi il existe une exécution acceptante ρ_1 de $\mathcal{A}[I = J_1]$ sur u ou
   il existe une exécution acceptante ρ_2 de $\mathcal{A}[I = J_2]$ sur u
   ssi $u \in \mathcal{L}(\mathcal{A}, J_1)$ ou $u \in \mathcal{L}(\mathcal{A}, J_2)$
   ssi $u \in \mathcal{L}(\mathcal{A}, J_1) \cup \mathcal{L}(\mathcal{A}, J_2)$
   En effet il suffit, dans le sens direct de prendre $\rho_1 = \rho_2 = \rho$, et dans le sens indirect de prendre $\rho = \rho_1$ ou $\rho = \rho_2$.
2. La preuve de cette égalité est similaire. Dans le sens indirect, on prendra cette fois $\rho = \rho_1 \cup \rho_2$: $\sigma(\Gamma) = \sigma_1(\Gamma_1) \cup \sigma_2(\Gamma_2)$ or $\sigma_1(\Gamma_1) \models J_1$ et $\sigma_2(\Gamma_2) \models J_2$ donc, d'après la proposition 4.1.2, $\sigma(\Gamma) \models J_1 \wedge J_2$. □

4.1.4 Q-DAGs

Nous allons maintenant introduire la notion de Q-DAG, un graphe acyclique dirigé étiqueté dans Q, puis montrer que les Q-DAG peuvent être utilisés de façon équivalente aux Q-forêts.

Définition 4.6 (*Q-DAG*)

Soit Q un ensemble fini. Un Q-DAG est un triplet (V, E, σ), où :
- (V, E) est un graphe acyclique dirigé,
- $\sigma : V \to Q$ est une fonction d'étiquetage,
- il existe une fonction de profondeur $\nu : V \to \mathbb{N}^*$ qui vérifie
  $$\forall y \in V \begin{cases} \text{si } \forall x \in V,\ (x,y) \notin E, \text{ alors } \nu(y) = 1 \text{ et} \\ \forall x \in V, \text{ si } (x,y) \in E, \text{ alors } \nu(y) = \nu(x) + 1 \end{cases}$$
- $(\sigma, \nu) : V \to Q \times \mathbb{N}^*$ est injective : il n'existe pas deux nœuds de même profondeur et de même étiquette.

On notera $\nu^{-1}(n) = \{x \in V \mid \nu(x) = n\}$, et on utilisera les mêmes notations que pour les Q-forêts.

Nous pouvons utiliser indifféremment les Q-forêts ou les Q-DAG pour les exécutions d'un automate alternant. En effet il est possible de passer aisément de l'un à l'autre par la construction expliquée dans la preuve de cette proposition.

Proposition 4.3

Un mot $u = u_0 u_1 \dots u_n \dots$ est accepté par un automate alternant $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ si et seulement si il existe un Q-DAG $\rho = (V, E, \sigma)$ tel que :
- *Les racines satisfont la condition initiale : $\sigma(\nu^{-1}(1)) \models I$,*
- *Les fils d'un nœud satisfont la fonction de transition :*
  $\forall x \in V,\ \sigma(E(x)) \models \delta(\sigma(x), u_{\nu(x)})$,
- *Tout chemin infini du Q-DAG ρ possède un nombre infini de nœuds étiquetés dans R.*

Démonstration. Soit $\rho = (V, E, \sigma)$ une Q-forêt. $\forall x \in V$, nous définissons la *mesure* du nœud x, notée $\mu(x)$, comme le plus petit entier N vérifiant la propriété suivante : sur toute branche infinie de ρ passant par x, il existe un descendant y de x étiqueté dans R tel que $\nu(y) \leq \nu(x) + N$. Si aucun entier N ne vérifie cette propriété, on pose $\mu(x) = \infty$. Dans le sous-arbre issu de x, $\mu(x)$ est la plus petite profondeur à partir de laquelle on a vu un nœud étiqueté dans R sur toutes les branches infinies. Si $\exists x \in V$ tel que $\mu(x) = \infty$, alors $\forall N \geq 0$, ρ contient une branche infinie dont aucun nœud n'est étiqueté dans R entre les profondeurs $\nu(x)$ et $\nu(x) + N$. Ainsi le sous-arbre restreint aux nœuds $\{y \in E^*(x) \mid \sigma(y) \notin R\}$ est un arbre infini à branchement fini, et le lemme de König (*une arbre infini dont les branchements sont finis possède une branche infinie*) nous permet d'affirmer que $\mu(x) = \infty$ si et seulement si il existe une branche infinie passant par x sans état répété après x. Ainsi ρ est acceptante si et seulement si $\forall x \in V$, $\mu(x)$ est fini.

$\boxed{\Rightarrow}$ Soit $\rho_0 = (V_0, E_0, \sigma_0)$ une Q-forêt, exécution acceptante de $\mathcal{A}$ sur u. Nous allons construire un Q-DAG qui respecte les conditions listées ci-dessus.

Modifions progressivement notre Q-forêt par récurrence. Pour tout entier $i \geq 0$, nous voulons construire une Q-forêt ρ_i, exécution acceptante de $\mathcal{A}$ sur u, telle que :

1. si $i \geq 1$, ρ_i est identique à ρ_{i-1} jusqu'à la profondeur i incluse,
2. deux nœuds de même étiquette et de même profondeur inférieure ou égale à i possèdent des sous-arbres identiques,
3. si $i \geq 1$, tout nœud x de profondeur inférieure ou égale à i vérifie $\mu_i(x) \leq \mu_{i-1}(x)$.

ρ_0 satisfait trivialement ces hypothèses (la profondeur d'un nœud est toujours supérieure à 1). Soit $i \geq 1$. Supposons que nous avons construit une Q-forêt ρ_{i-1} correspondant à ces contraintes : la Q-forêt ρ_i sera obtenue à partir de ρ_{i-1} de la façon suivante : pour tout état q de Q, soit $A_{i,q}$ l'ensemble des nœuds de V_{i-1} d'étiquette q et de profondeur i. Choisissons un élément x de $A_{i,q}$ de mesure minimale, et remplaçons le sous-arbre issu de chaque nœud de $A_{i,q}$ par une copie du sous-arbre issu de x.

Vérifions maintenant que nos conditions sont vérifiées pour ρ_i :

1. Il est clair que la construction de ρ_i ne modifie pas les nœuds de profondeur inférieure à i.
2. Soient x_1 et x_2 deux nœuds de ρ_i d'étiquette (q, j) avec $j \leq i$. Si $j < i$, alors d'après le point précédent ces nœuds sont des nœuds de ρ_{i-1}, et par hypothèse de récurrence leurs sous-arbres sont identiques dans ρ_{i-1}. Leurs sous-arbres dans ρ_i sont nécessairement identiques, car si l'un d'eux a subi une modification l'autre a nécessairement subi la même modification. Si $j = i$, alors ces deux nœuds sont dans $A_{i,q}$, et possèdent donc par construction le même sous-arbre dans ρ_i.
3. Soit x un nœud de ρ_i d'étiquette (q, j) avec $j \leq i$. Le point 1 assure que x est aussi un nœud de ρ_{i-1}. Si $\mu_{i-1}(x) \leq i - j$, alors sur toutes les branches infinies passant par x il existe un descendant de x de profondeur inférieure à i dont l'étiquette est dans R. Comme tous les descendants de x de profondeur inférieure à i dans ρ_{i-1} sont conservés dans ρ_i, on a $\mu_i(x) = \mu_{i-1}(x)$. Sinon, si $\mu_{i-1}(x) > i - j$, alors soit

$A = \{y \in E^*_{i-1}(x) \mid \nu_{i-1}(y) = i\}$: $\mu_{i-1}(x) = i - j + \max_{y \in A} \mu_{i-1}(y)$. Or le point 1 nous assure que $A = \{y \in E^*_i(x) \mid \nu_i(y) = i\}$, et $\mu_i(x) = i - j + \max_{y \in A} \mu_i(y)$, et, de plus, par construction de ρ_i, $\forall y \in A$, $\mu_i(y) \leq \mu_{i-1}(y)$, donc on en déduit que $\mu_i(x) \leq \mu_{i-1}(x)$.

Le point 1 implique que la suite $(\rho_i)_{i\geq 0}$ converge vers une valeur limite $\rho = (V, E, \sigma)$, exécution de $\mathcal{A}$ sur u, telle que 2 nœuds de même étiquette et de même profondeur possèdent des sous-arbres identiques.

Montrons que ρ est acceptante : soit x un nœud de ρ. $\forall i \geq 1$, ρ est identique à ρ_i jusqu'à la profondeur i incluse. Ainsi x apparaît dans $\rho_{\nu(x)}$. La condition 3 assure que $\mu(x) \leq \mu_{\nu(x)}(x)$. Ainsi $\forall x \in V$, $\mu(x)$ est fini : ρ est acceptante.

Il suffit maintenant de transformer ρ en un Q-DAG, en fusionnant successivement les nœuds de même profondeur et de même étiquette (ce qui ne pose pas de problème puisque les sous-arbres de deux tels nœuds sont dorénavant identiques). Il est clair qu'on obtient de cette façon un Q-DAG de la définition 4.6, et ρ étant une exécution acceptante de $\mathcal{A}$ sur u, et ce Q-DAG vérifie les propriétés de la proposition 4.3.

$\boxed{\Leftarrow}$ Soit $\rho_0 = (V_0, E_0, \sigma_0)$ un Q-DAG vérifiant les propriétés de la proposition 4.3 : nous allons éclater ce graphe en une Q-forêt, exécution acceptante de $\mathcal{A}$ sur u, en construisant par récurrence une suite $(\rho_i)_{i\geq 0}$ de graphes. Pour tout $i \geq 1$, à partir du graphe ρ_{i-1} nous définissons $\rho_i = (V_i, E_i, \sigma_i)$ de la façon suivante :

- Soit $A = \{x \in V_{i-1} \mid \nu_{i-1}(x) = i\}$,
- $V_i = V_{i-1} \backslash A \cup \{x_y \mid x \in A \text{ et } (y, x) \in E_{i-1}\}$.
- $E_i = E_{i-1} \backslash \{(x, y) \in E_{i-1} \mid x \in A \text{ ou } y \in A\} \cup \{(y, x_y) \mid x \in A \text{ et } (y, x) \in E_{i-1}\}$ $\cup \{(x_y, z) \mid x \in A \text{ et } (y, x), (x, z) \in E_{i-1}\}$
- σ_i est identique à σ_{i-1} sur $V_{i-1} \backslash A$, et si $x \in A$ et $(y, x) \in E_{i-1}$, $\sigma_i(x_y) = \sigma_{i-1}(x)$.

Le passage de ρ_{i-1} à ρ_i consiste en fait à remplacer chaque nœud d'incidence k à la profondeur i par k nœuds d'incidence 1, de même étiquette et de même sous-DAG, afin de transformer progressivement notre Q-DAG en une Q-forêt. La suite de graphes définie ainsi converge vers une Q-forêt, exécution acceptante de $\mathcal{A}$ sur u. □

Remarque.

> Par la suite, nous utiliserons indifféremment une Q-forêt ou un Q-DAG pour représenter une exécution d'un automate alternant. Ce choix permettra de simplifier les preuves en utilisant directement la structure de données la plus appropriée.

4.2 Exemple

Nous présentons dans cette section un exemple d'automate alternant. Il s'agit de l'automate obtenu à partir de la formule $\theta_1 = \neg(\mathsf{G}\,\mathsf{F}\,p \rightarrow \mathsf{G}(r \rightarrow \mathsf{F}\,g))$ par l'algorithme présenté ultérieurement. Il est présenté sur la figure 4.1.

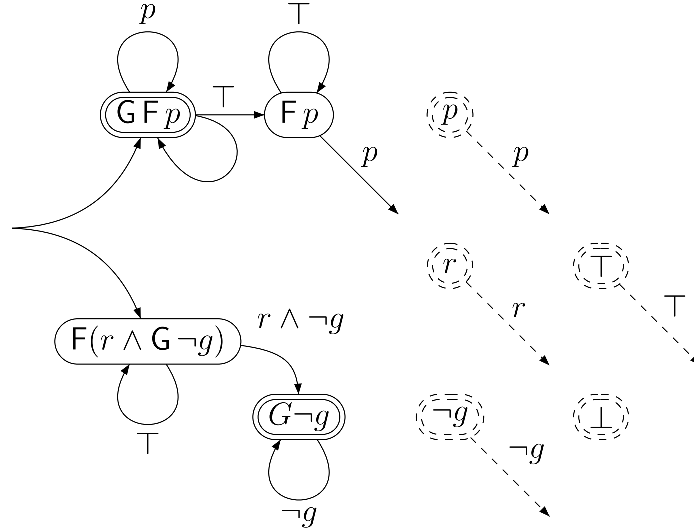


FIG. 4.1 – Automate alternant $\mathcal{A}_{\theta_1}$.

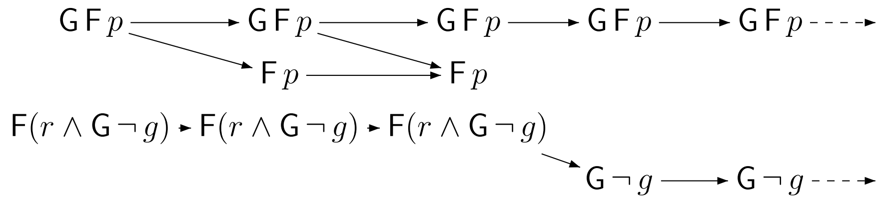


FIG. 4.2 – Exécution acceptante de $\mathcal{A}_{\theta_1}$ sur le mot $u = \emptyset\{r, g\}\{p, r\}\{p\}^\omega$

Les états répétés sont notés dans un double cercle. Les états non accessibles sont notés dans un cercle pointillé. Deux flèches de même origine représentent une conjonction dans la fonction de transition.

Sur cet automate, nous avons, à titre d'exemple :

- $I = \mathsf{G}\,\mathsf{F}\,p \;\wedge\; \mathsf{F}(r \wedge \mathsf{G}\neg g)$,
- $\delta(p, \{p\}) = \top$,
- $\delta(\mathsf{G}\,\mathsf{F}\,p, \{\}) = \mathsf{G}\,\mathsf{F}\,p \;\wedge\; \mathsf{F}\,p$.

Ou, avec les notations optimisées de la section 7.1.3,

- $I = \{\{\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}(r \wedge \mathsf{G}\neg g)\}\}$,
- $\delta(p) = \{(\Sigma_p, \emptyset)\}$,
- $\delta(\mathsf{G}\,\mathsf{F}\,p) = \{(\Sigma_p, \{\mathsf{G}\,\mathsf{F}\,p\}), (\Sigma, \{\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}\,p\})\}$.

La figure 4.2 est une exécution acceptante de $\mathcal{A}_{\theta_1}$ sur le mot $u = \emptyset\{r, g\}\{p, r\}\{p\}^\omega$.

4.3 De LTL aux automates alternants très faibles

4.3.1 Construction

Dans cette section nous exposons comment, à partir d'une formule LTL φ de taille temporelle $|\varphi|$, construire un automate alternant très faible $\mathcal{A}_\varphi$ de même langage, avec au plus $|\varphi|$ états.

Définition 4.7 ($\varphi \to \mathcal{A}_\varphi$)

Soit φ une formule LTL sur l'ensemble de propositions atomiques AP, en forme normale négative. L'automate $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, R)$ est défini ainsi :
- Q est l'ensemble des sous-formules temporelles de φ,
- $\Sigma = 2^{\mathrm{AP}}$,
- $I = \varphi \in \mathcal{B}^+(Q)$,
- R est l'ensemble des éléments de Q qui ne sont pas de la forme $\psi_1 \mathsf{U} \psi_2$,
- δ est la restriction à $Q \times \Sigma$ de la fonction suivante, définie sur $\mathcal{B}^+(Q) \times \Sigma$:

$$\forall a \in \Sigma, \begin{cases} \delta(\top, a) = \top \\ \delta(\bot, a) = \bot \\ \delta(p, a) = \top \text{ si } p \in a, \bot \text{ sinon} \\ \delta(\neg p, a) = \top \text{ si } p \notin a, \bot \text{ sinon} \\ \delta(\psi_1 \vee \psi_2, a) = \delta(\psi_1, a) \vee \delta(\psi_2, a) \\ \delta(\psi_1 \wedge \psi_2, a) = \delta(\psi_1, a) \wedge \delta(\psi_2, a) \\ \delta(\mathsf{X}\,\psi, a) = \psi \in \mathcal{B}^+(Q) \\ \delta(\psi_1 \mathsf{U} \psi_2, a) = \delta(\psi_2, a) \vee (\delta(\psi_1, a) \wedge (\psi_1 \mathsf{U} \psi_2)) \\ \delta(\psi_1 \mathsf{R} \psi_2, a) = \delta(\psi_2, a) \wedge (\delta(\psi_1, a) \vee (\psi_1 \mathsf{R} \psi_2)) \end{cases}$$

Remarques.

L'ensemble Q des états, contrairement aux définitions classiques, est restreint aux sous-formules *temporelles* de φ. Toutefois, pour pouvoir traiter récursivement le cas de toutes les sous-formules de φ, on étend pour la construction δ à $\mathcal{B}^+(Q) \times \Sigma$ (toute sous-formule de φ peut être vue comme une formule de $\mathcal{B}^+(Q)$).

C'est cette dernière remarque qui justifie qu'on puisse définir I et $\delta(\mathsf{X}\,\psi, a)$ comme ils l'ont été.

Les éléments $\top$ et $\bot$ sont ambigus. À gauche de l'égalité, ce sont des sous-formules temporelles de φ. À droite, ce sont par contre des éléments de $\mathcal{B}^+(Q)$, le premier étant satisfait par tout sous-ensemble de Q, le deuxième n'étant satisfait par aucun d'entre eux.

Proposition 4.4

Pour toute formule LTL φ, l'automate $\mathcal{A}_\varphi$ est un automate alternant très faible.

Démonstration. Il suffit de prendre l'ordre partiel "être une sous-formule de" pour se convaincre aisément de cette affirmation. ☐

4.3.2 Preuve de l'équivalence

Nous allons maintenant prouver que $\mathcal{A}_\varphi$ reconnaît le langage de la formule φ, en démontrant au préalable deux lemmes intermédiaires.

Notation : $\forall \mathcal{L} \subseteq \Sigma^\omega$, $\forall a \in \Sigma$, on définit $a \cdot \mathcal{L} = \{a.u \mid u \in \mathcal{L}\}$.

Lemme 4.5

Pour toute sous-formule ψ de φ, $\mathcal{L}(\mathcal{A}_\varphi, \psi) = \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi, a))$

Démonstration. Nous allons prouver cette égalité par induction sur ψ.

- Si $\psi = \top$, $\mathcal{L}(\mathcal{A}_\varphi, \psi) = \Sigma^\omega$ (la forêt vide est une exécution valide, étant donné que l'ensemble $\emptyset$ des étiquettes de ses racines satisfait $\top$).
  D'où $\mathcal{L}(\mathcal{A}_\varphi, \top) = \Sigma^\omega = \bigcup_{a\in\Sigma} a \cdot \Sigma^\omega = \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\top, a))$

- Si $\psi = \bot$, $\mathcal{L}(\mathcal{A}_\varphi, \psi) = \emptyset$ (il n'existe aucune exécution valide, étant donné que $\forall Q' \subseteq Q$, $Q' \not\models \bot$).
  D'où $\mathcal{L}(\mathcal{A}_\varphi, \bot) = \emptyset = \bigcup_{a\in\Sigma} a \cdot \emptyset = \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\bot, a))$

- Si ψ est une formule temporelle différente de $\top$ et $\bot$,

  $\boxed{\subseteq}$ Si $u = a.v \in \mathcal{L}(\mathcal{A}_\varphi, \psi)$, il existe Q-forêt ρ, exécution acceptante de $\mathcal{A}_\varphi[I = \psi]$ sur u. $\psi \in \sigma(\Gamma)$, donc il existe une racine x de ρ d'étiquette ψ. Par définition, $\sigma(E(x)) \models \delta(\psi, \alpha)$. Si on considère la sous-Q-forêt ρ' de ρ ayant pour racines les fils de x, celle-ci est une exécution acceptante de $\mathcal{A}_\varphi[I = \delta(\psi, \alpha)]$ sur le mot v, ce qui signifie que $v \in \mathcal{L}(\delta(\psi, \alpha))$. Ainsi, $u \in a \cdot \mathcal{L}(\delta(\psi, \alpha))$.

  $\boxed{\supseteq}$ Réciproquement, prenons $u = a.v$ avec $a \in \Sigma$ et $v \in \mathcal{L}(\delta(\psi, \alpha))$. Il existe une Q-forêt ρ', exécution acceptante de $\mathcal{A}_\varphi[I = \delta(\psi, \alpha)]$ sur le mot v, avec $\sigma(\Gamma') \models \delta(\psi, \alpha)$. Si on ajoute un nœud étiqueté par ψ, ayant pour fils les racines de ρ', et devenant l'unique racine d'un Q-arbre ρ, alors ρ est une exécution acceptante de $\mathcal{A}_\varphi[I = \psi]$ sur u. Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \psi)$.

  La propriété est vérifiée si ψ est une formule temporelle.

- Si $\psi = \psi_1 \vee \psi_2$,

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \psi_1 \vee \psi_2) &= \mathcal{L}(\mathcal{A}_\varphi, \psi_1) \cup \mathcal{L}(\mathcal{A}_\varphi, \psi_2) && \textit{(lemme 4.2.1)}\\
&= \left(\bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a))\right) \cup \left(\bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_2, a))\right) && \textit{(hyp. d'induction)}\\
&= \bigcup_{a\in\Sigma} a \cdot (\mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a)) \cup \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_2, a))\\
&= \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a) \vee \delta(\psi_2, a)) && \textit{(lemme 4.2.1)}\\
&= \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1 \vee \psi_2, a)) && \textit{(définition de } \delta)
\end{aligned}$$

– Si $\psi = \psi_1 \wedge \psi_2$,

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \psi_1 \wedge \psi_2) &= \mathcal{L}(\mathcal{A}_\varphi, \psi_1) \cap \mathcal{L}(\mathcal{A}_\varphi, \psi_2) && \textit{(lemme 4.2.2)} \\
&= \left(\bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a))\right) \cap \left(\bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_2, a))\right) && \textit{(hyp. d'induction)} \\
&= \bigcup_{a\in\Sigma} a \cdot (\mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a)) \cap \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_2, a))) \\
&= \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, a) \wedge \delta(\psi_2, a)) && \textit{(lemme 4.2.2)} \\
&= \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1 \wedge \psi_2, a)) && \textit{(définition de } \delta)
\end{aligned}$$

□

Lemme 4.6

Pour toute sous-formule ψ de φ, $\mathcal{L}(\mathcal{A}_\varphi, \psi) = \mathcal{L}(\psi)$.

Démonstration. Nous allons prouver cette affirmation par induction sur ψ :

– $\mathcal{L}(\mathcal{A}_\varphi, \top) = \Sigma^\omega = \mathcal{L}(\top)$

– $\mathcal{L}(\mathcal{A}_\varphi, \top) = \emptyset = \mathcal{L}(\bot)$

– $\forall a \in \Sigma$, $\delta(p, a) = \top$ si $p \in a$, $\bot$ sinon.

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, p) &= \bigcup_{a\in\Sigma, p\in a} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \top) \cup \bigcup_{a\in\Sigma, p\notin a} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \bot) && \textit{(lemme 4.5)} \\
&= \{a \in \Sigma \mid p \in a\} \cdot \mathcal{L}(\top) \cup \{a \in \Sigma \mid p \notin a\} \cdot \mathcal{L}(\bot) && \textit{(hypothèse d'induction)} \\
&= \mathcal{L}(p) && \textit{(sémantique de LTL)}
\end{aligned}$$

– $\forall a \in \Sigma$, $\delta(\neg p, a) = \top$ si $p \notin a$, $\bot$ sinon.

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \neg p) &= \bigcup_{a\in\Sigma, p\notin a} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \top) \cup \bigcup_{a\in\Sigma, p\in a} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \bot) && \textit{(lemme 4.5)} \\
&= \{a \in \Sigma \mid p \notin a\} \cdot \mathcal{L}(\top) \cup \{a \in \Sigma \mid p \in a\} \cdot \mathcal{L}(\bot) && \textit{(hypothèse d'induction)} \\
&= \mathcal{L}(\neg p) && \textit{(sémantique de LTL)}
\end{aligned}$$

– $$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \psi_1 \vee \psi_2) &= \mathcal{L}(\mathcal{A}_\varphi, \psi_1) \cup \mathcal{L}(\mathcal{A}_\varphi, \psi_2) && \textit{(lemme 4.2.1)} \\
&= \mathcal{L}(\psi_1) \cup \mathcal{L}(\psi_2) && \textit{(hypothèse d'induction)} \\
&= \mathcal{L}(\psi_1 \vee \psi_2) && \textit{(sémantique de LTL)}
\end{aligned}$$

– $$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \psi_1 \wedge \psi_2) &= \mathcal{L}(\mathcal{A}_\varphi, \psi_1) \cap \mathcal{L}(\mathcal{A}_\varphi, \psi_2) && \textit{(lemme 4.2.2)} \\
&= \mathcal{L}(\psi_1) \cap \mathcal{L}(\psi_2) && \textit{(hypothèse d'induction)} \\
&= \mathcal{L}(\psi_1 \wedge \psi_2) && \textit{(sémantique de LTL)}
\end{aligned}$$

– $\forall a \in \Sigma$, $\delta(\mathsf{X}\,\psi, a) = \psi$

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \mathsf{X}\,\psi) &= \bigcup_{a\in\Sigma} a \cdot \mathcal{L}(\mathcal{A}_\varphi, \psi) && \textit{(lemme 4.5)} \\
&= \Sigma \cdot \mathcal{L}(\psi) && \textit{(hypothèse d'induction)} \\
&= \mathcal{L}(\mathsf{X}\,\psi) && \textit{(sémantique de LTL)}
\end{aligned}$$

– $\psi = \psi_1 \mathsf{U} \psi_2$: $\forall a \in \Sigma$, $\delta(\psi_1 \mathsf{U} \psi_2, a) = \delta(\psi_2, a) \vee (\delta(\psi_1, a) \wedge (\psi_1 \mathsf{U} \psi_2))$

$\boxed{\subseteq}$ Considérons une Q-forêt ρ, exécution acceptante de $\mathcal{A}_\varphi[I = \psi]$ sur le mot $u = u_1 u_2 \ldots$: il existe une racine x_1 de ρ d'étiquette ψ.

Si $\sigma(E(x)) \models \delta(\psi_1, u_1) \wedge (\psi_1 \mathsf{U} \psi_2)$, alors un des fils x_2 de x_1 a pour étiquette ψ. Ce nœud peut lui-même avoir un fils x_3 d'étiquette ψ et ainsi de suite, mais ceci ne peut pas se produire indéfiniment, car $\psi \notin R$ donc ρ ne peut contenir de branche infinie dont les nœuds sont tous étiquetés par ψ.

Ainsi x_1 est le premier élément d'une succession de n nœuds $x_1, \ldots, x_n$ tous étiquetés par ψ tels que (voir la figure 4.3) :

- $\forall 1 \leq i < n$, $\sigma(E(x_i) \backslash \{x_{i+1}\}) \models \Delta(\psi_1, u_i)$, donc la Q-forêt obtenue en tronquant ρ en prenant $E(x_i) \backslash \{x_{i+1}\}$ comme racines est une exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_1, u_i)]$ sur le mot $u_{i+1} u_{i+2} \ldots$ Ainsi, $u_{i+1} u_{i+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_1, u_i))$ donc, en utilisant le lemme 4.5, $u_i u_{i+1} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \psi_1)$. Par hypothèse d'induction, on en déduit que $u_i u_{i+1} \ldots \models \psi_1$.
- $\sigma(E(x_n)) \models \Delta(\psi_2, u_n)$, donc la Q-forêt obtenue en tronquant ρ en prenant $E(x_n)$ comme racines est une exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_2, u_n)]$ sur le mot $u_{n+1} u_{n+2} \ldots$ Ainsi, $u_{n+1} u_{n+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_2, u_n))$ donc, en utilisant le lemme 4.5, $u_n u_{n+1} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \psi_2)$. Par hypothèse d'induction, on en déduit $u_n u_{n+1} \ldots \models \psi_2$.

La sémantique de LTL nous permet de conclure que $u \models \psi$.

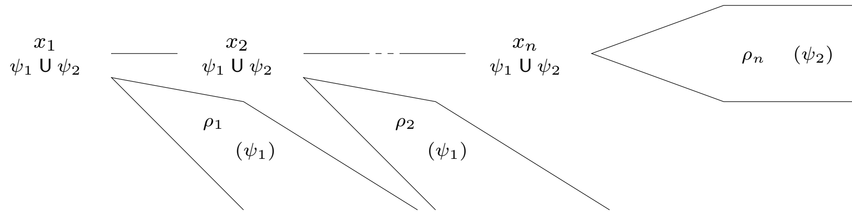


FIG. 4.3 – Exécution pour $\psi_1 \mathsf{U} \psi_2$

$\boxed{\supseteq}$ Réciproquement, si $u \models \psi$, par la sémantique de LTL il existe un entier n tel que $\forall 1 \leq i < n$, $u_i u_{i+1} \ldots \models \psi_1$ et $u_n u_{n+1} \ldots \models \psi_2$. En utilisant l'hypothèse d'induction puis le lemme 4.5, on obtient $\forall 1 \leq i < n$, $u_{i+1} u_{i+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_1, u_i))$ et $u_{n+1} u_{n+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_2, u_n))$.

Ainsi $\forall 1 \leq i < n$ on peut construire une Q-forêt $\rho_i = (V_i, E_i, \sigma_i)$, exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_1, u_i)]$ sur $u_{i+1} u_{i+2} \ldots$ et une Q-forêt $\rho_n = (V_n, E_n, \sigma_n)$, exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_2, u_n)]$ sur $u_{n+1} u_{n+2} \ldots$ Créons n nouveaux nœuds $x_1, \ldots, x_n$ et une nouvelle Q-forêt $\rho = (V, E, \sigma)$ définie ainsi :

- $V = \bigcup_{1 \leq i \leq n} V_i \cup \{x_i \mid 1 \leq i \leq n\}$,
- $E = \bigcup_{1 \leq i \leq n} E_i \cup \bigcup_{1 \leq i \leq n} (\{x_i\} \times \Gamma_i) \cup \bigcup_{1 \leq i < n} (x_i, x_{i+1})$,
- $\forall 1 \leq i \leq n$, $\sigma(x_i) = \psi$ et $\forall x \in V_i$, $\sigma(x) = \sigma_i(x)$.

Le résultat est visible sur la figure 4.3.

x_1 est l'unique racine de la Q-forêt ρ, dont on voit aisément qu'elle est une exécution de $\mathcal{A}_\varphi[I = \psi]$ sur u. De plus cette exécution est acceptante, car toutes ses branches infinies finissent comme des branches infinies des ρ_i. D'où $u \in \mathcal{L}(\mathcal{A}_\varphi, \psi)$.

– $\psi = \psi_1 \,\mathsf{R}\, \psi_2$: $\forall a \in \Sigma$, $\delta(\psi_1 \,\mathsf{R}\, \psi_2, a) = \delta(\psi_2, a) \wedge (\delta(\psi_1, a) \vee (\psi_1 \,\mathsf{R}\, \psi_2))$

$\boxed{\subseteq}$ Considérons une Q-forêt ρ, exécution acceptante de $\mathcal{A}_\varphi[I = \psi]$ sur le mot $u = u_1u_2\dots$: elle possède une racine x_1 d'étiquette ψ.

Si $\sigma(E(x)) \models \delta(\psi_2, a) \wedge (\psi_1 \,\mathsf{R}\, \psi_2)$, alors un des fils x_2 de x_1 a pour étiquette ψ. Ce nœud peut lui-même avoir un fils x_3 d'étiquette ψ et ainsi de suite. Cette fois-ci, il est possible que ρ contienne une branche infinie dont les nœuds sont tous étiquetés par ψ.

Ainsi x_1 est le premier élément d'une succession finie ou infinie de n ($n \in \mathbb{N}^* \cup \{\omega\}$) nœuds $x_1, x_2, \dots$ tous étiquetés par ψ tels que (voir figures 4.4 et 4.5) :

- $\forall i \in \mathbb{N}$, si $1 \leq i \leq n$, $\sigma(E(x_i)\backslash\{x_{i+1}\}) \models \Delta(\psi_2, u_i)$, donc la Q-forêt obtenue en tronquant ρ en prenant $E(x_i)\backslash\{x_{i+1}\}$ comme racines est une exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_2, u_i)]$ sur le mot $u_{i+1}u_{i+2}\dots$ Ainsi, $u_{i+1}u_{i+2}\dots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_2, u_i))$ donc, en utilisant le lemme 4.5, $u_iu_{i+1}\dots \in \mathcal{L}(\mathcal{A}_\varphi, \psi_2)$. Par hypothèse d'induction, on en déduit que $u_iu_{i+1}\dots \models \psi_2$.
- si n est fini, $\sigma(E(x_n) \models \Delta(\psi_1, u_n)$, donc la Q-forêt obtenue en tronquant ρ en prenant $E(x_n)$ comme racines est une exécution acceptante de $\mathcal{A}_\varphi[I = \Delta(\psi_1, u_n)]$ sur le mot $u_{n+1}u_{n+2}\dots$ Ainsi, $u_{n+1}u_{n+2}\dots \in \mathcal{L}(\mathcal{A}_\varphi, \Delta(\psi_1, u_n))$ donc, en utilisant le lemme 4.5, $u_nu_{n+1}\dots \in \mathcal{L}(\mathcal{A}_\varphi, \psi_1)$. Par hypothèse d'induction, on en déduit que $u_nu_{n+1}\dots \models \psi_1$.

La sémantique de LTL nous permet de conclure que $u \models \psi$.

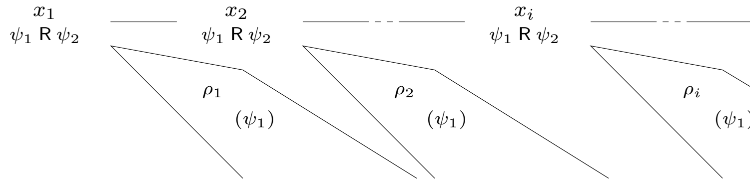


FIG. 4.4 – Exécution pour $\psi_1 \,\mathsf{R}\, \psi_2$ – $n = \omega$

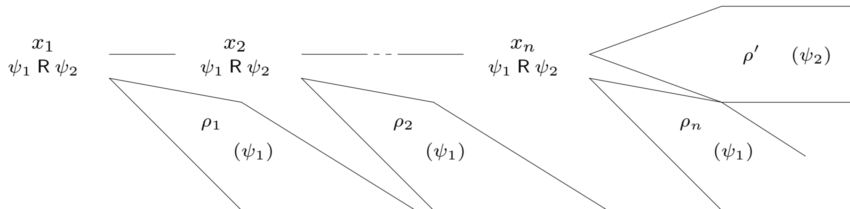


FIG. 4.5 – Exécution pour $\psi_1 \,\mathsf{R}\, \psi_2$ – $n \in \mathbb{N}^*$

$\boxed{\supseteq}$ Réciproquement, si $u \models \psi$, par la sémantique de LTL il existe $n \in \mathbb{N}^* \cup \{\omega\}$ tel que $\forall i \in \mathbb{N}$, si $1 \leq i \leq n$, $u_i u_{i+1} \ldots \models \psi_2$ et si n est fini, $u_n u_{n+1} \ldots \models \psi_1$. En utilisant l'hypothèse d'induction puis le lemme 4.5, on obtient $\forall i \in \mathbb{N}$, si $1 \leq i \leq n$, $u_{i+1}u_{i+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_2, u_i))$ et si n est fini, $u_{n+1}u_{n+2} \ldots \in \mathcal{L}(\mathcal{A}_\varphi, \delta(\psi_1, u_n))$.
Ainsi $\forall i \in \mathbb{N}$, si $1 \leq i \leq n$ on peut construire une Q-forêt $\rho_i = (V_i, E_i, \sigma_i)$, exécution acceptante de $\mathcal{A}_\varphi[I = \delta(\psi_2, u_i)]$ sur $u_{i+1}u_{i+2} \ldots$ et si n est fini on peut construire une Q-forêt $\rho_{n+1} = (V', E', \sigma')$, exécution acceptante de $\mathcal{A}_\varphi[I = \delta(\psi_1, u_n)]$ sur $u_{n+1}u_{n+2} \ldots$ Créons n nouveaux nœuds $x_1, x_2, \ldots$ et une nouvelle Q-forêt $\rho = (V, E, \sigma)$ définie ainsi :

- si $n = \omega$, $V = \bigcup_{i \geq 1} V_i \cup \{x_i \mid i \geq 1\}$,
  sinon $V = \bigcup_{1 \leq i \leq n+1} V_i \cup \{x_i \mid 1 \leq i \leq n\}$,
- si $n = \omega$, $E = \bigcup_{i \geq 1} E_i \cup \bigcup_{i \geq 1} (\{x_i\} \times \Gamma_i) \cup \bigcup_{i \geq 1} (x_i, x_{i+1})$,
  sinon $E = \bigcup_{1 \leq i \leq n+1} E_i \cup \bigcup_{1 \leq i \leq n} (\{x_i\} \times \Gamma_i) \cup \{x_n\} \times \Gamma_{n+1} \cup \bigcup_{1 \leq i < n} (x_i, x_{i+1})$,
- si $n = \omega$, $\forall i \geq 1$, $\sigma(x_i) = \psi$ et $\forall x \in V_i$, $\sigma(x) = \sigma_i(x)$,
  sinon, $\forall 1 \leq i \leq n$, $\sigma(x_i) = \psi$ et $\forall 1 \leq i \leq n+1$, $\forall x \in V_i$, $\sigma(x) = \sigma_i(x)$.

Le résultat est visible sur les figures 4.4 et 4.5.

x_1 est l'unique racine de la Q-forêt ρ, dont on voit aisément qu'elle est une exécution de $\mathcal{A}_\varphi[I = \psi]$ sur u. De plus cette exécution est acceptante, car toutes ses branches infinies finissent comme des branches infinies des ρ_i, exceptée, dans le cas où $n = \omega$, la branche des x_i, étiquetée par $\psi \in R$. D'où $u \in \mathcal{L}(\mathcal{A}_\varphi, \psi)$. □

Nous pouvons maintenant aisément conclure : l'automate $\mathcal{A}_\varphi$ est bien celui que nous avions promis.

Théorème 4.7

Pour toute formule LTL φ, $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$.

Démonstration. Il suffit d'appliquer le lemme précédent à $\psi = \varphi$. □

4.3.3 Complexité

Pour toute formule LTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U} \psi_2$, l'automate $\mathcal{A}_\varphi$ possède au maximum n états et $n - r$ états répétés.

Chapitre 5

Automates de Büchi généralisés

Nous sommes pour l'instant capables de générer un automate alternant très faible reconnaissant le langage d'une formule LTL. Cet automate ne peut pas être utilisé en model-checking, et doit donc être transformé en un automate plus classique.

La construction usuelle pour transformer un automate alternant en un automate de Büchi produit un automate qui est trop gros : pour un automate alternant à n états, l'automate de Büchi aura $2^n \times 2^n$ états. Si ce facteur exponentiel ne peut pas être évité, il peut toutefois être amélioré, si l'automate alternant est très faible. Nous nous permettons d'utiliser cette optimisation, profitant du fait que l'automate alternant que nous obtenons à partir d'une formule LTL est toujours très faible.

Notre construction utilise les automates de Büchi généralisés. Dans ce chapitre nous définirons les automates de Büchi généralisés, et exposerons la construction que nous proposons pour générer un tel automate à partir d'un automate alternant très faible.

5.1 Définition

Les automates de Büchi généralisés que nous utilisons généralisent les automates de Büchi sur deux points. Tout d'abord, ils utilisent plusieurs conditions d'acceptation, une exécution acceptante devant satisfaire toutes les conditions d'acceptation. D'autre part, les conditions d'acceptation portent sur les transitions, et non sur les états de l'automate comme c'est l'usage.

Cette généralisation nous permet de générer des automates plus petits et plus simples à partir d'un automate alternant très faible.

Définition 5.1 (*automate de Büchi généralisé*)

Un automate de Büchi généralisé est un quintuplet $\mathcal{G} = (Q, \Sigma, T, I, \mathcal{T})$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $T \subseteq Q \times \Sigma \times Q$ est la fonction de *transition*,
- $I \subseteq Q$ est l'ensemble des *états initiaux*,
- $\mathcal{T} = \{T_1, \ldots, T_r\}$ est l'ensemble des *tables d'acceptation* où $\forall 1 \leq j \leq r$, $T_j \subseteq T$.

Soit $u = u_1 u_2 \ldots \in \Sigma^\omega$. Une *exécution* ρ de $\mathcal{G}$ sur u est une suite infinie $q_0, q_1, \ldots$ d'éléments de Q telle que :
- le premier état est initial : $q_0 \in I$,
- on passe d'un état au suivant en suivant la fonction de transition : $\forall i \geq 1$, $(q_{i-1}, u_i, q_i) \in T$.

Une exécution ρ est *acceptante* si, de plus, elle utilise une infinité de transitions (q_{i-1}, u_i, q_i) dans chacune des tables T_j, $1 \leq j \leq r$.
Le *langage* $\mathcal{L}(\mathcal{G})$ d'un automate de Büchi généralisé $\mathcal{G}$ est l'ensemble des mots de Σ^ω sur lesquels il existe une exécution acceptante de $\mathcal{G}$.

5.2 Exemple

Nous présentons dans cette section un exemple d'automate de Büchi généralisé. Il s'agit de l'automate obtenu à partir de l'automate alternant $\mathcal{A}_{\theta_1}$ par l'algorithme présenté ultérieurement. Il est présenté sur la figure 5.1

Sur cet exemple, $\mathcal{T} = \{T_1, T_2\}$. Les transitions de T_1, qui correspondent à l'état $\mathsf{F}(q \wedge \mathsf{G}\neg r)$, sont notées en pointillés, et les transitions de T_2, qui correspondent à l'état $\mathsf{F}\,p$, sont notées en gras. Une exécution acceptante doit utiliser une infinité de transitions en pointillés et une infinité de transitions en gras.

La figure 5.2 présente le même automate après simplification par les méthodes présentées dans la section 7.2.

5.3 Des automates alternants très faibles aux automates de Büchi généralisés

5.3.1 Construction

Dans cette section nous exposons comment, à partir d'un automate alternant très faible $\mathcal{A}$ avec n états, construire un automate de Büchi généralisé $\mathcal{G}_\mathcal{A}$ de même langage, avec au plus 2^n états et n tables d'acceptation.

L'idée de cette construction est la suivante : on peut transformer une exécution de $\mathcal{A}$ en une exécution d'un automate ayant pour états les sous-ensembles de Q, en groupant les états de même profondeur dans chaque exécution de $\mathcal{A}$. Le problème de cet "aplatissement"

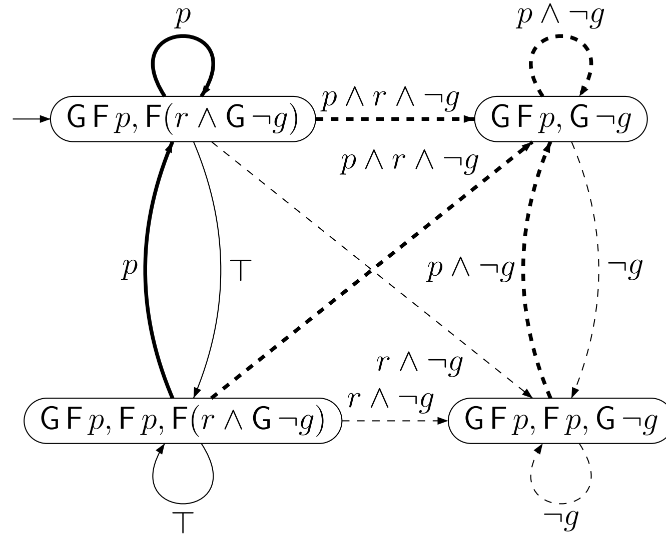


FIG. 5.1 – Automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}_{\theta_1}}$ avant simplification.

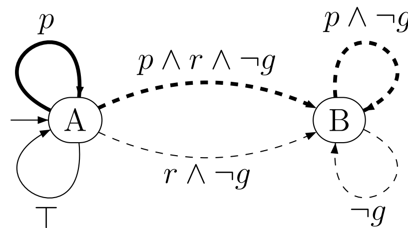


FIG. 5.2 – Automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}_{\theta_1}}$ après simplification.

d'une Q-forêt en une séquence sur 2^Q, est de traduire les conditions d'acceptation de $\mathcal{A}$. La solution que nous avons adoptée est la suivante : nous imposons que l'automate de Büchi généralisé répète infiniment souvent des transitions qui nous permettent de garantir qu'on peut construire une exécution de $\mathcal{A}$ qui n'utilise pas, à certaines profondeurs, d'arête dont les deux extrémités sont étiquetées par un état non répété.

Définition 5.2 ($\mathcal{A} \to \mathcal{G}_{\mathcal{A}}$)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible.
L'automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}} = (Q', \Sigma, T, I', \mathcal{T})$ est défini comme suit :
- $Q' = 2^Q$,
- $T = \{(X, a, X') \mid \forall q \in X,\ X' \models \delta(q, a)\}$,
- $I' = \{X \subseteq Q \mid X \models I\}$,
- $\mathcal{T} = \{T_q \mid q \in Q \backslash R\}$ où $T_q = \{(X, a, X') \in T \mid q \notin X$ ou $X' \backslash \{q\} \models \delta(q, a)$.

Remarque.

On peut imposer dans les définitions de T et de I' d'utiliser uniquement des sous-ensembles *minimaux* de Q qui satisfont respectivement δ et I , c'est-à-dire des ensembles $X \subseteq Q$ tels que, par exemple, $X \models I$ et $\forall X' \subsetneq X,\ X' \not\models I$.

Cette restriction ne change pas le langage accepté par l'automate, et permet de réduire fortement le nombre de transitions et le nombre d'états initiaux de l'automate $\mathcal{G}_{\mathcal{A}}$, et donc de simplifier les calculs. Toutefois, nous utiliserons dans le chapitre 7 un autre système qui possède des effets similaires, c'est pourquoi nous n'insistons pas sur ce sujet pour l'instant.

5.3.2 Preuve de l'équivalence

Nous allons prouver ici que $\mathcal{G}_{\mathcal{A}}$ possède le même langage que $\mathcal{A}$. Nous allons utiliser dans cette preuve le fait que $\mathcal{A}$ est très faible, ce qui permet d'utiliser le lemme suivant, qui est crucial :

Lemme 5.1

Soit $\mathcal{A}$ un automate alternant très faible. Soit ρ une exécution de $\mathcal{A}$ sur un mot u : la suite des étiquettes de toute branche infinie de ρ est ultimement constante. Ainsi une branche contenant une infinité de nœuds étiquetés dans R si et seulement si sa limite est dans R, et elle en contient un nombre fini si et seulement si sa limite est dans $Q \backslash R$.

Démonstration. Puisque $\mathcal{A}$ est très faible, il existe un ordre partiel sur Q tel que la suite des étiquettes de cette branche infinie est décroissante. Une suite décroissante dans un ensemble fini est ultimement constante. Les deux propriétés énoncées sont alors immédiates. □

Théorème 5.2

Pour tout automate alternant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G}_{\mathcal{A}})$.

Démonstration.

$\boxed{\subseteq}$ Soit $u = u_1u_2\ldots \in \mathcal{L}(\mathcal{A})$. Il existe un Q-DAG $\rho = (V, E, \sigma)$, exécution acceptante de $\mathcal{A}$ sur le mot u.

$\forall i \geq 1$, posons $X_i = \sigma(\nu^{-1}(i))$. Nous allons prouver que $\rho' = X_1, X_2, \ldots$ est une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u.

- D'après la proposition 4.3, $\sigma(\nu^{-1}(1)) \models I$, donc $X_1 \models I$, c'est-à-dire $X_1 \in I'$. La condition initiale est satisfaite.
- Soient $i \geq 1$ et $A = \nu^{-1}(i)$. Par définition de la profondeur ν, $\nu^{-1}(i+1) = \bigcup_{x\in A} E(x)$. Ainsi $X_{i+1} = \sigma(\nu^{-1}(i+1)) = \bigcup_{x\in A} \sigma(E(x))$. D'après la proposition 4.3, pour tout nœud $x \in V$, $\sigma(E(x)) \models \delta(\sigma(x), u_{\nu(x)})$. Ainsi, si $\forall x \in A$, on pose $A_x = \sigma(E(x))$, alors $X_{i+1} = \bigcup_{x\in A} A_x$ où $A_x \subseteq Q$ et $A_x \models \delta(\sigma(x), u_i)$. D'après la proposition 4.1.1, on en déduit que $\forall q \in X_i$, $X_{i+1} \models \delta(q, u_i)$, donc ρ' respecte la fonction de transition de $\mathcal{G}_\mathcal{A}$ sur u.
- Montrons enfin que les conditions d'acceptation sont satisfaites : soient $i \geq 0$ et $q \in Q\backslash R$. Nous voulons montrer qu'il existe $j \geq i$ tel que $(X_j, u_j, X_{j+1}) \in T_q$. Prenons pour j le plus petit entier supérieur à i tel qu'il n'y a pas dans ρ d'arête entre deux nœuds de profondeurs j et $j+1$ tous deux étiquetés par q (cet entier existe, sinon, ρ étant un Q-DAG, il existerait une branche infinie de ρ ultimement étiquetée par q, et ρ ne vérifierait pas les hypothèses de la proposition 4.3). S'il n'existe pas de nœud de profondeur j étiqueté par q, alors $q \notin X_j$ et $(X_j, u_j, X_{j+1}) \in T_q$. Sinon, soit x le nœud de profondeur j étiqueté par q (ρ est un Q-DAG donc (σ, ν) est injective). Le nœud x n'a pas de successeur étiqueté par q, donc $\sigma(E(x)) \subseteq X_{j+1}\backslash\{q\}$ et $\sigma(E(x)) \models \delta(q, u_j)$, donc d'après la proposition 4.1.1, $X_{j+1}\backslash\{q\} \models \delta(q, u_j)$, d'où $(X_j, u_j, X_{j+1}) \in T_q$.

Ainsi, ρ' est une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u : $u \in \mathcal{L}(\mathcal{G}_\mathcal{A})$.

$\boxed{\supseteq}$ Soit $u = u_1u_2\ldots \in \mathcal{L}(\mathcal{A})$. Il existe une exécution acceptante $\rho' = X_1, X_2, \ldots$ de $\mathcal{G}_\mathcal{A}$ sur u.

Soit $V = \{(q,i) \mid i \geq 1 \text{ et } q \in X_i\}$. V sera l'ensemble des nœuds d'un Q-DAG ρ que nous allons construire, de telle sorte que ρ soit une exécution acceptante de $\mathcal{A}$ sur u. Pour tout nœud (q,i) de V, nous définissons $\sigma((q,i)) = q$, et par construction $\nu((q,i)) = i$.

Par définition de T, $\forall(q,i) \in V$, $q \in X_i$ donc $X_{i+1} \models \delta(q, u_i)$. Si $X_{i+1}\backslash\{q\} \models \delta(q, u_i)$, alors on pose $A_{q,i} = X_{i+1}\backslash\{q\}$, et dans les autres cas on pose $A_{q,i} = X_{i+1}$. Définissons maintenant $E = \{((q,i),(q',i+1)) \mid (q,i) \in V \text{ et } q' \in A_{q,i}\}$ (pour chaque nœud $x = (q,i)$ de V, $E(x) = A_{q,i} \times \{i+1\}$).

Soit $\rho = (V, E, \sigma)$. Il est clair que ρ est un Q-DAG.

- $X_0 \in I'$ donc $X_0 = \sigma(\nu^{-1}(1)) \models I$. Les racines de ρ satisfont la condition initiale.
- Soit $x = (q,i) \in V$. $A_{q,i} = \sigma(E(x)) \models \delta(q, u_i) = \delta(\sigma(x), u_\nu(x))$. Les fils d'un nœud satisfont la fonction de transition.
- Supposons que ρ contienne une branche infinie qui ne possède qu'un nombre fini de nœuds étiquetés dans R. D'après le lemme 5.1, la suite des étiquettes des nœuds de cette branche est ultimement constante, et aurait pour limite $q \in Q\backslash R$. Mais ρ' est

acceptante : cette exécution utilise une infinité de transitions dans T_q, et au niveau de chacune de ces transitions, nous avons défini E de telle sorte qu'il n'y ait pas d'arête entre deux nœuds étiquetés par q. Ceci contredit notre hypothèse, et donc toute branche infinie de ρ possède un nombre infini de nœuds étiquetés dans R.

Ainsi ρ vérifie toutes les conditions de la proposition 4.3, donc $u \in \mathcal{L}(\mathcal{A})$ □

5.3.3 Complexité

Pour toute automate alternant très faible $\mathcal{A}$ avec n états et $n - r$ états répétés, l'automate $\mathcal{G}_{\mathcal{A}}$ possède au maximum 2^n états et r tables d'acceptation.

Ainsi pour toute formule LTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U} \psi_2$, nous sommes capables de générer un automate de Büchi généralisé de même langage, avec au maximum 2^n états et r tables d'acceptation.

Chapitre 6

Model-checking pour LTL

Nous savons maintenant générer à partir de toute formule φ de taille temporelle n un automate de Büchi généralisé avec au maximum 2^n états et n tables d'acceptation, reconnaissant le langage de φ.

Or les algorithmes de model-checking usuels utilisent des automates de Büchi classiques, et non des automates de Büchi généralisés. Deux solutions sont donc possibles : transformer l'automate de Büchi généralisé en automate de Büchi classique de même langage, ou bien effectuer le model-checking sur l'automate de Büchi généralisé lui-même.

6.1 Des automates de Büchi généralisés aux automates de Büchi classiques

Dans cette section nous exposons comment, à partir d'un automate de Büchi généralisé $\mathcal{G}$ avec n états et r tables d'acceptation, construire un automate de Büchi classique $\mathcal{B}_{\mathcal{G}}$, avec au plus $n \times (r+1)$ états.

Cet algorithme est classique, voici une idée de son fonctionnement : il suffit de considérer cet automate de Büchi généralisé comme un automate de Büchi classique sans conditions d'acceptation, et de s'assurer par un autre moyen que les conditions d'acceptation sont satisfaites. Pour cela, il suffit de construire un automate de Büchi dont les exécutions visitent infiniment souvent chaque table d'acceptation, et d'en faire le produit synchronisé avec $\mathcal{G}$. Nous reviendrons sur cette idée par la suite.

6.1.1 Construction

Définition 6.1 ($\mathcal{G} \to \mathcal{B}_\mathcal{G}$)

Soit $\mathcal{G} = (Q, \Sigma, T, I, \mathcal{T})$ un automate de Büchi généralisé, avec $\mathcal{T} = \{T_1, \ldots, T_r\}$. L'automate de Büchi $\mathcal{B}_\mathcal{G} = (Q', \Sigma, \delta, I', R)$ est défini comme suit :

- $Q' = Q \times \{0, \ldots, r\}$,
- $\texttt{next} : \{0, \ldots, r\} \times (Q \times \Sigma \times Q) \to \{0, \ldots, r\}$ est défini par :
$$\begin{cases} \texttt{next}(j,t) = \max\{j \leq i \leq r \mid \forall j < k \leq i,\ t \in T_k\} \text{ si } j \neq r \text{ et} \\ \texttt{next}(r,t) = \max\{0 \leq i \leq r \mid \forall 0 < k \leq i,\ t \in T_k\}, \end{cases}$$
- $\delta((q,j),a) = \{(q',j') \mid (q,a,q') \in T \text{ et } j' = \texttt{next}(j,(q,a,q'))\}$,
- $I' = I \times \{0\}$,
- $R = Q \times \{r\}$.

Remarque.

Supposons qu'après avoir lu n lettres, l'automate $\mathcal{B}_\mathcal{G}$ soit dans l'état (q,j). Par construction, q est l'état de $\mathcal{G}$ correspondant, et on assure que depuis le dernier passage dans un état de R, une transition a été vue dans chacune des tables T_i, $1 \leq i \leq j$. Ainsi entre deux états de l'ensemble R, on a visité au moins une fois chacune des tables d'acceptation de $\mathcal{T}$.

6.1.2 Automates d'acceptation

Nous allons présenter ici quelques automates d'acceptation qui peuvent être utilisés pour transformer un automate de Büchi généralisé $\mathcal{G}$ en un automate de Büchi classique $\mathcal{B}$, et justifier le choix que nous avons fait.

Un automate d'acceptation est en fait un automate de Büchi sur l'alphabet $\mathcal{T}$ des tables d'acceptation de $\mathcal{G}$, qui accepte tous les mots contenant chaque T_j une infinité de fois.

Plusieurs automates peuvent être utilisés en fonction des résultats souhaités. Quelques possibilités sont données ici à titre d'exemple, avec une représentation de l'automate correspondant pour $\mathcal{T} = \{T_1, T_2, T_3\}$.

- La première possibilité est celle de la figure 6.1. Les états, au nombre de 2^r, sont les parties de $\{1, \ldots, r\}$. L'état courant indique quelles conditions d'acceptation ont été satisfaites depuis le dernier passage par l'état $\{1, \ldots, r\}$. Lors d'une transition d'un état à un autre, on ajoute à l'état toutes les conditions d'acceptation satisfaites. L'état $\{1, \ldots, r\}$ est répété, car entre deux passages par cet état toute les tables d'acceptation ont été visitées.
  Cet automate présente l'avantage de prendre en compte toutes les tables d'acceptation visitées au fur et à mesure du calcul, dans n'importe quel ordre, et permet ainsi de minimiser le contre-exemple obtenu dans le processus de model-checking. Cependant il contient beaucoup d'états et l'automate de Büchi obtenu devient vite assez gros.
- La deuxième possibilité est celle de la figure 6.2, à gauche. Les états sont au nombre de $r + 1$, et l'état courant i indique que les tables T_1 à T_i ont été visitées depuis le

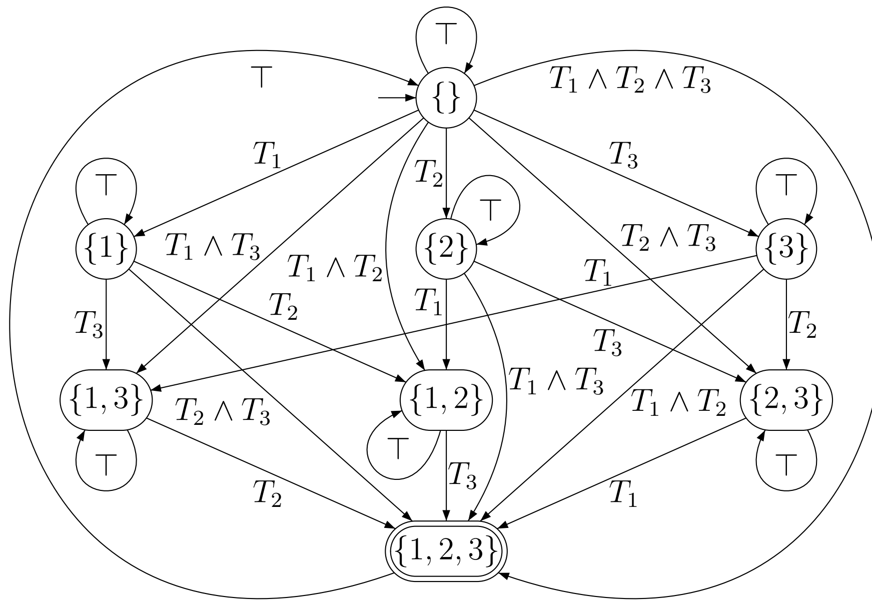


FIG. 6.1 – Automates d'acceptation (1)

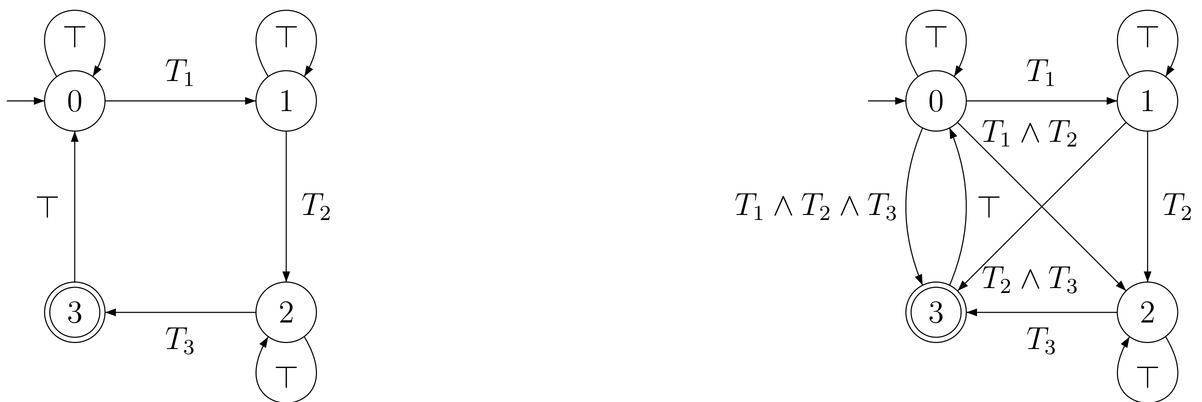


FIG. 6.2 – Automates d'acceptation (2)

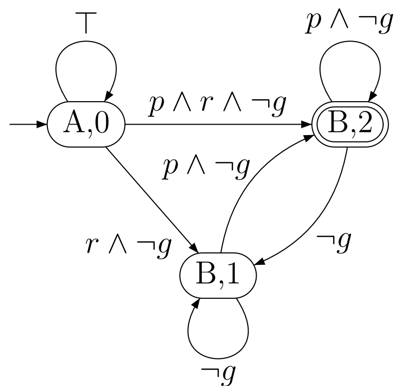


FIG. 6.3 – Automate de Büchi $\mathcal{B}_{\mathcal{G}_{\mathcal{A}_{\theta_1}}}$.

dernier passage par l'état r. Dans chaque état $i < r$, on fait une transition vers l'état $i+1$ si la transition est dans l'ensemble T_{i+1}. L'état r est répété, car entre deux passages par cet état toutes les tables d'acceptation ont été visitées.
Cet automate présente l'avantage de posséder un nombre minimal d'états et de transitions, ce qui se répercute sur la taille de l'automate de Büchi obtenu. Par contre la taille du contre-exemple sera fortement augmentée car on impose de visiter les conditions d'acceptation T_1 à T_r une par une, dans l'ordre.
- La troisième possibilité est celle de la figure 6.2, à droite. Les états sont les mêmes que dans l'automate précédent, mais cette fois on autorise la passage de l'état i à l'état j si les tables de $i+1$ à j sont visitées par la transition courante.
Cet automate est intermédiaire entre les automates précédents, il contient un nombre minimal d'états mais peut prendre en compte plusieurs conditions d'acceptation simultanément, afin de réduire la taille du contre-exemple. Il continue cependant à imposer un ordre arbitraire sur les tables.
- Pour chacune des possibilités, on peut encore améliorer chaque automate en prenant pour l'état répété les mêmes transitions que pour l'état initial, au lieu de la simple transition étiquetée par $\top$.
En appliquant cette méthode au troisième automate présenté, on retrouve l'automate que nous avons utilisé dans notre présentation et dans notre implémentation, dont la construction est présentée dans la définition 6.1. Il nous semble un bon compromis entre la taille de l'automate de Büchi obtenu et la taille du contre-exemple obtenu en model-checking.

La figure 6.3 présente, après simplification, l'automate de Büchi obtenu en appliquant l'algorithme précédent à l'automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}_{\theta_1}}$ présenté dans la figure 5.2.

6.1.3 Preuve de l'équivalence

Nous allons prouver ici que $\mathcal{B}_\mathcal{G}$ possède le même langage que $\mathcal{G}$.

Théorème 6.1

Pour tout automate de Büchi généralisé $\mathcal{G}$, $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{B}_\mathcal{G})$.

Démonstration.

$\boxed{\subseteq}$ Soit $u = u_1u_2\ldots \in \mathcal{L}(\mathcal{G})$: il existe une exécution acceptante $\rho = q_1, q_2, \ldots$ de $\mathcal{G}$ sur u. Créons une suite $(j_i)_{i\geq 1}$ définie ainsi : $j_1 = 0$ et $\forall i \geq 1$, $j_{i+1} = \texttt{next}(j_i, (q_i, u_i, q_{i+1}))$. La suite $\rho' = (q_1, j_1), (q_2, j_2), \ldots$ est clairement une exécution de $\mathcal{B}_\mathcal{G}$ sur u.

Supposons que ρ' n'est pas acceptante : il existe un entier N tel que $\forall i \geq N$, $j_i \neq r$. Or, par définition de la fonction `next`, $j_i \neq r \Rightarrow j_{i+1} \geq j_i$. Ainsi la suite $(j_i)_{i\geq N}$ est croissante, et majorée strictement par r. Elle atteint donc une limite $k < r$. Ainsi une fois cette limite atteinte, aucune transition (q_i, u_i, q_{i+1}) ne peut être dans l'ensemble T_{k+1}. Ceci contredit le fait que ρ est une exécution acceptante. Ainsi ρ' est une exécution acceptante de $\mathcal{B}_\mathcal{G}$ sur u, donc $u \in \mathcal{L}(\mathcal{B}_\mathcal{G})$.

$\boxed{\supseteq}$ Soit $u = u_1u_2 \ldots \in \mathcal{L}(\mathcal{B}_\mathcal{G})$: il existe une exécution acceptante $\rho' = (q_1, j_1), (q_2, j_2), \ldots$ de $\mathcal{B}_\mathcal{G}$ sur u. La suite $\rho = q_1, q_2, \ldots$ est clairement une exécution de $\mathcal{G}$ sur u.

La définition de `next` permet d'affirmer la propriété suivante : $\forall i \neq i'$, si $j_i = j_{i'} = r$, l'exécution ρ utilise au moins une transition dans chacune des tables de $\mathcal{T}$ entre les positions i et i'. Ainsi, puisque ρ' est acceptante, une infinité de j_i sont égaux à r, et donc ρ utilise une infinité de transitions dans chacune des tables de $\mathcal{T}$. ρ est acceptante, et donc $u \in \mathcal{L}(\mathcal{G})$. $\square$

6.1.4 Complexité

Pour tout automate de Büchi généralisé $\mathcal{G}$ avec n états et r conditions d'acceptation, l'automate $\mathcal{B}_\mathcal{G}$ possède au maximum $n \times (r + 1)$ états.

Ainsi pour toute formule LTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U} \psi_2$, nous sommes capables de générer un automate de Büchi de même langage, avec au maximum $(r + 1) \times 2^n$ états.

6.2 Automates de Büchi généralisés et model-checking

Si la solution proposée ci-dessus est commode, puisqu'elle permet de réutiliser des outils de model-checking existants, elle possède des problèmes, évoqués dans le paragraphe 6.1.2. De plus, le nombre d'états est multiplié par le nombre de tables d'acceptation. Il serait en fait plus efficace d'effectuer le model-checking directement à partir de l'automate de Büchi généralisé.

6.2.1 Principe

La méthode est simple : de manière similaire à l'algorithme classique, on effectue le produit synchronisé de l'automate de Büchi généralisé avec le modèle. L'automate obtenu reste un automate de Büchi généralisé, sur lequel on peut effectuer le test du vide.

Le test du vide d'un automate de Büchi généralisé est simple à effectuer : comme avec les automates de Büchi classiques, il faut calculer les composantes fortement connexes, puis chercher une composante fortement connexe comportant au moins une transition dans chacune des tables d'acceptation.

6.2.2 Complexité

Le test du vide d'un automate de Büchi généralisé avec n états et r tables d'acceptation se fait en temps linéaire, soit $\mathcal{O}(n^2 \times r \times |\Sigma|)$ dans le pire des cas.

En effet cet automate comporte au pire $n^2 \times |\Sigma|$ transitions, et chacune des r tables d'acceptation comporte ainsi au plus $n^2 \times |\Sigma|$ éléments. Le calcul des composantes fortement connexes avec l'algorithme de Tarjan [Tar72] se fait en temps $\mathcal{O}(n^2 \times |\Sigma|)$, et il faut ensuite pour toutes les transitions au sein de chaque composante fortement connexe faire la liste des tables d'acceptation satisfaites, et on retrouve le résultat $\mathcal{O}(n^2 \times r \times |\Sigma|)$ annoncé.

Chapitre 7

Implémentation

Nous allons dans ce chapitre parler plus en détail de l'implémentation que nous avons réalisée à partir de cet algorithme. Nous approfondirons plusieurs points : l'optimisation de la représentation des automates, leur simplification, et les résultats expérimentaux que nous avons obtenus.

7.1 Optimisation des structures de données

Il y a plusieurs raisons importantes pour lesquelles nous avons cherché à optimiser la représentation de nos structures de données. La première est la volonté de faire des représentations compactes qui occupent moins de place en mémoire, l'espace mémoire nécessaire aux calculs étant en effet une ressource critique dans le model-checking. La deuxième raison est que des structures appropriées permettent de faire des calculs plus simples et plus rapides, et donc de réduire le temps de calcul.

7.1.1 L'étiquetage des transitions

Le premier domaine de simplification est celui des étiquettes des transitions. Tous les automates que nous avons présentés ont des transitions étiquetées dans $\Sigma = 2^{\mathrm{AP}}$, les étiquettes sont donc des sous-ensembles de l'ensemble des propositions atomiques de la formule LTL.

Prenons un exemple : considérons une formule LTL φ sur AP utilisant n propositions atomiques. Soit $p \in \mathrm{AP}$. Au cours de la génération de l'automate alternant, l'état p possède 2^n transitions sortantes : la moitié d'entre elles vers $\top$ (si $p \in a$), l'autre vers $\bot$ (si $p \notin a$).

Cette multiplication inutile de toutes les transitions se répercute sur l'automate de Büchi généralisé, et pour chaque automate elle est très coûteuse en mémoire, et aussi en temps de calcul puisqu'il est souvent nécessaire de parcourir l'ensemble des transitions issues d'un état.

Notre idée est la suivante : nous voulons grouper au maximum les transitions d'étiquettes différentes qui vont d'un même état à un même état. Pour cela l'étiquette d'une transition sera dans 2^{Σ}.

Une observation plus attentive nous a fait remarquer que tous les éléments de 2^{Σ} ne sont pas utilisés : en effet, on s'intéresse uniquement aux propriétés $p \in a$ ou $p \notin a$. Si on note $\Sigma_p = \{a \in \Sigma \mid p \in a\}$, les éléments de 2^{Σ} qui nous sont utiles peuvent être obtenus par intersection finie des ensembles Σ, Σ_p et $\Sigma_{\neg p} = \Sigma \backslash \Sigma_p$ pour $p \in \mathrm{AP}$.

Ceci nous permet d'optimiser la représentation en mémoire des étiquettes des transitions : une étiquette $\alpha \in 2^{\Sigma}$ sera représentée par une paire $(P, N) \in \Sigma \times \Sigma$, avec

$$\alpha = \Sigma \cap \left(\bigcap_{p \in P} \Sigma_p \right) \cap \left(\bigcap_{p \in N} \Sigma_{\neg p} \right)$$

Ainsi en représentant un élément de Σ comme une suite de $|\mathrm{AP}|$ bits, les éléments de 2^{Σ} qui nous intéressent peuvent être représentés sur $2 \times |\mathrm{AP}|$ bits, et l'état p de l'automate alternant aura uniquement deux transitions : l'une étiquetée par Σ_p vers $\top$, l'autre étiquetée par $\Sigma_{\neg p}$ vers $\bot$. Cette représentation, en plus d'être compacte, permet d'accélérer les calculs : les deux outils dont nous avons besoin, à savoir l'intersection et le test d'inclusion, peuvent être effectuée par les opérations bit à bit : & (et), | (ou), ! (non).

$$\begin{array}{rcl} (P, N) \cap (P', N') & = & (P \mid P', N \mid N') \\ (P, N) \subseteq (P', N') & \iff & P' \ \& \ (!\ P) = N' \ \& \ (!\ N) = 0 \end{array}$$

Ceci facilite aussi la lisibilité des résultats : l'étiquette $(\{p\}, \{q\}) = \Sigma_p \cap \Sigma_{\neg q}$ sera affichée sous la forme $p \wedge \neg q$.

Par ailleurs, la fonction de transition n'étant maintenant définie que sur certains éléments de 2^{Σ}, il est plus efficace de la présenter différemment. Ainsi, par exemple, pour l'automate de Büchi généralisé, on définira $\delta : Q \to 2^{2^{\Sigma} \times Q}$. Les transitions issues d'un état seront donc un ensemble de paires (étiquette, état d'arrivée).

7.1.2 Transitions des automates alternants

Un autre point majeur peut être amélioré dans la représentation des automates alternants. Les transitions de ceux-ci utilisent des combinaisons booléennes positives sur Q. Or celles-ci possèdent deux inconvénients.

Le premier inconvénient est la proposition 4.1.1 : si un ensemble satisfait une formule, alors tout sur-ensemble la satisfait aussi. Ceci multiplie énormément le nombre de transitions de l'automate de Büchi généralisé. Une solution pour remédier à ce problème est d'utiliser uniquement les sous-ensembles minimaux de Q qui satisfont la fonction de transition.

Le deuxième inconvénient est qu'il est ridicule de devoir tester à chaque fois que c'est nécessaire l'ensemble des parties de Q qui satisfont la fonction de transition, et que c'est encore plus coûteux s'il faut sélectionner parmi celles-ci les ensembles minimaux.

La solution intuitive à ce problème est d'utiliser, au lieu d'une combinaison booléenne positive sur Q, la liste des sous-ensembles minimaux qui satisfont cette formule (un élément de 2^{2^Q}). C'est ce que nous faisons approximativement.

Voici ce que deviennent les formules que nous utilisions :

$$\begin{array}{rl} \bot & \text{devient } \emptyset \\ \top & \text{devient } \{\emptyset\} \\ q & \text{devient } \{\{q\}\} \\ J_1 \vee J_2 & \text{devient } J_1 \cup J_2 \\ J_1 \wedge J_2 & \text{devient } J_1 \otimes J_2 = \{X_1 \cup X_2 \mid X_1 \in J_1 \text{ et } X_2 \in J_2\} \end{array}$$

7.1.3 Définitions et constructions

Utilisant ces deux optimisations, nous pouvons reformuler les définitions des automates et les constructions de l'algorithme. Toutes les preuves peuvent être adaptées (parfois avec quelques difficultés) à ce formalisme. Toutefois nous ne détaillerons pas les preuves ici, nous préférons vous renvoyer à l'article qui les détaille : [GO01].

Définition 7.1 (*automate alternant optimisé*)

Un *automate alternant* est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $\delta : Q \to 2^{(2^\Sigma \times 2^Q)}$ est la *fonction de transition*,
- $I \in 2^{2^Q}$ est la *condition initiale*,
- $R \subseteq Q$ est l'ensemble des *états répétés*.

Une *exécution* de l'automate $\mathcal{A}$ sur le mot u est une Q-forêt $\rho = (V, E, \sigma)$ telle que :
- Les racines satisfont la condition initiale : $\sigma(\Gamma) \in I$,
- Les fils d'un nœud satisfont la fonction de transition : $\forall x \in V,\ \exists \alpha \in 2^\Sigma,\ u_i \in \alpha$ et $(\alpha, \sigma(E(x))) \in \delta(\sigma(x))$.

Remarque.

Pour $J_1, J_2 \in 2^{2^\Sigma \times 2^Q}$ on définit $J_1 \otimes J_2 \in 2^{2^\Sigma \times 2^Q}$ par
$J_1 \otimes J_2 = \{(\alpha_1 \cap \alpha_2, X_1 \cup X_2) \mid (\alpha_1, X_1) \in J_1 \text{ et } (\alpha_2, X_2) \in J_2\}$,
Pour une formule LTL ψ on définit $\overline{\psi} \in 2^{2^Q}$ par :
- $\overline{\psi} = \{\{\psi\}\}$ si ψ est une formule temporelle,
- $\overline{\psi_1 \wedge \psi_2} = \{X_1 \cup X_2 \mid X_1 \in \overline{\psi_1} \text{ et } X_2 \in \overline{\psi_2}\}$,
- $\overline{\psi_1 \vee \psi_2} = \overline{\psi_1} \cup \overline{\psi_2}$.

Définition 7.2 (*$\varphi \rightarrow \mathcal{A}_\varphi$ optimisé*)

Soit φ une formule LTL sur l'ensemble de propositions atomiques AP.
L'automate $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, R)$ est défini ainsi :
- Q est l'ensemble des sous-formules temporelles de φ,
- $\Sigma = 2^{\mathrm{AP}}$,
- $I = \overline{\varphi}$,
- R est l'ensemble des éléments de Q qui ne sont pas de la forme $\psi_1 \,\mathsf{U}\, \psi_2$,
- δ est la restriction à Q de la fonction suivante, définie sur $\mathcal{B}^+(Q)$:

$$\begin{cases} \delta(\bot) = \emptyset \\ \delta(\top) = \{(\Sigma, \emptyset)\} \\ \delta(p) = \{(\Sigma_p, \emptyset)\} \\ \delta(\neg p) = \{(\Sigma_{\neg p}, \emptyset)\} \\ \delta(\psi_1 \vee \psi_2) = \delta(\psi_1) \cup \delta(\psi_2) \\ \delta(\psi_1 \wedge \psi_2) = \delta(\psi_1) \otimes \delta(\psi_2) \\ \delta(\mathsf{X}\, \psi) = \{(\Sigma, X) \mid X \in \overline{\psi}\} \\ \delta(\psi_1 \,\mathsf{U}\, \psi_2) = \delta(\psi_2) \;\cup (\delta(\psi_1) \otimes \{(\Sigma, \{\psi_1 \,\mathsf{U}\, \psi_2\})\}) \\ \delta(\psi_1 \,\mathsf{R}\, \psi_2) = \delta(\psi_2) \;\otimes (\delta(\psi_1) \cup \{(\Sigma, \{\psi_1 \,\mathsf{R}\, \psi_2\})\}) \end{cases}$$

Définition 7.3 (***automate de Büchi généralisé optimisé***)

Un automate de Büchi généralisé est un quintuplet $\mathcal{G} = (Q, \Sigma, T, I, \mathcal{T})$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $T \subseteq Q \times 2^\Sigma \times Q$ est la fonction de *transition*,
- $I \subseteq Q$ est l'ensemble des *états initiaux*,
- $\mathcal{T} = \{T_1, \ldots, T_r\}$ est l'ensemble des *tables d'acceptation* où $\forall 1 \leq j \leq r$, $T_i \subseteq T$.

Soit $u = u_1 u_2 \ldots \in \Sigma^\omega$. Une *exécution* ρ de $\mathcal{G}$ sur u est une suite infinie $q_0, q_1, \ldots$ d'éléments de Q telle que :
- le premier état est initial : $q_0 \in I$,
- on passe d'un état au suivant en suivant la fonction de transition :
  $\forall i \geq 1$, $\exists \alpha \in 2^\Sigma$, $u_i \in \alpha$ et $(\alpha_i, q_i) \in \delta(q_{i-1})$.

Définition 7.4 (*$\mathcal{A} \rightarrow \mathcal{G}_\mathcal{A}$ optimisé*)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible.
L'automate de Büchi généralisé $\mathcal{G}_\mathcal{A} = (Q', \Sigma, T, I', \mathcal{T})$ est défini comme suit :
- $Q' = 2^Q$,
- $T = \{(X, \alpha, X') \mid (\alpha, X') = \bigotimes_{q \in X} \delta(q)\}$,
- $I' = I$,
- $\mathcal{T} = \{T_q \mid q \in Q \backslash R\}$ où
  $T_q = \{(X, \alpha, X') \in T \mid q \notin X' \text{ ou } \exists(\beta, X'') \in \delta(q),\ \alpha \subseteq \beta,\ X'' \subseteq X' \backslash \{q\}\}$

Remarques.

Vous aurez peut-être remarqué que dans cette construction, lors de la définition de T_q, nous avons écrit $q \notin X'$ et non $q \notin X$. Ce changement est délibéré : la construction reste correcte, et nous avons observé qu'elle permet d'obtenir en pratique des automates qui se simplifient mieux et sont plus petits au final.

Contrairement au cas des autres constructions optimisées, ici la preuve du fait que les deux automates ont le même langage est difficile à adapter à partir de celle exposée dans le chapitre 5. C'est pourquoi nous allons exposer cette preuve en détail, à la suite du théorème 7.4.

Définition 7.5 (*automate de Büchi optimisé*)

Un automate de Büchi est un quintuplet $\mathcal{B} = (Q, \Sigma, \delta, I, R)$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $\delta : Q \to 2^{2^{\Sigma} \times Q}$ est la fonction de *transition*,
- $I \subseteq Q$ est l'ensemble des *états initiaux*,
- $R \subseteq Q$ est l'ensemble des *états répétés*,

Soit $u = u_1 u_2 \ldots \in \Sigma^\omega$. Une *exécution* ρ de $\mathcal{B}$ sur u est une suite infinie $q_0, q_1, \ldots$ d'éléments de Q telle que :
- le premier état est initial : $q_0 \in I$,
- on passe d'un état au suivant en suivant la fonction de transition : $\forall i \geq 1$, $\exists \alpha \in 2^\Sigma$, $u_i \in \alpha$ et $(\alpha_i, q_i) \in \delta(q_{i-1})$.

7.2 Simplification des automates

La simplification des automates est un des points clefs de la supériorité de notre algorithme devant ses concurrents. Simplifier à tour de rôle chaque automate avant de passer à l'étape suivante permet de gagner un facteur majeur en espace mémoire et en temps de calcul.

Pour simplifier un automate, nous appliquons successivement trois règles de simplification, en itérant jusqu'à ce que plus aucune de ces simplification n'ait d'effet. Ces règles de simplification sont les suivantes :
- Simplification des transitions
- Simplification des états
- Utilisation des composantes fortement connexes

7.2.1 Simplification des transitions

Si une transition $t_2 = (q, \alpha_2, q_2)$ est plus *contraignante* qu'une transition $t_1 = (q, \alpha_1, q_1)$, alors la transition t_2 peut être supprimée.

- Pour un automate alternant très faible, t_2 est plus contraignante que t_1 si $\alpha_2 \subseteq \alpha_1$ et $q_1 \subseteq q_2$.
- Pour un automate de Büchi généralisé, t_2 est plus contraignante que t_1 si $\alpha_2 \subseteq \alpha_1$, $q_1 = q_2$ et $\forall T_j \in \mathcal{T}$, $t_2 \in T_j \Rightarrow t_1 \in T_j$.
- Pour un automate de Büchi, t_2 est plus contraignante que t_1 si $\alpha_2 \subseteq \alpha_1$ et $q_1 = q_2$.

Dans le cas particulier d'un automate de Büchi généralisé directement issu d'un automate alternant très faible par notre construction, les états sont des ensembles, et on peut utiliser $q_1 \subseteq q_2$ au lieu de $q_1 = q_2$ dans la condition ci-dessus. Toutefois ceci devient faux dès que des états ont été fusionnés par l'étape de simplification des états de la section 7.2.2.

Proposition 7.1 (*automate alternant*)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible (défini sous sa version optimisée). Si $\exists q \in Q\ \exists (\alpha_1, X_1), (\alpha_2, X_2) \in \delta(q)$ tels que $\alpha_2 \subseteq \alpha_1$ et $X_1 \subseteq X_2$ mais $\alpha_1 \neq \alpha_2$ ou $X_2 \neq X_1$, alors on peut supprimer (α_2, X_2) de $\delta(q)$.

Démonstration. Soit $\mathcal{A}'$ l'automate modifié, après suppression de la transition. Nous allons prouver que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

$\boxed{\subseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$ et soit $\rho = (V, E, \sigma)$ une exécution acceptante de $\mathcal{A}$ sur u. Pour tout nœud $x \in V$ d'étiquette q tel que $u_{\nu(x)} \in \alpha_2$ et $\sigma(E(x)) = X_2$, supprimons tous les fils de x d'étiquette $q \in X_2 \backslash X_1$ et leurs sous-arbres. On a alors $\sigma(E(x)) = X_1$ et $u_{\nu(x)} \in \alpha_1$ car $\alpha_2 \subseteq \alpha_1$. La Q-forêt ρ' ainsi obtenue reste une exécution acceptante de $\mathcal{A}$ sur u et n'utilise plus la transition (q, α_2, X_2) : c'est une exécution acceptante de $\mathcal{A}'$ sur u, et $u \in \mathcal{L}(\mathcal{A}')$.

$\boxed{\supseteq}$ Toute exécution acceptante de l'automate $\mathcal{A}'$ sur un mot u est aussi une exécution acceptante de $\mathcal{A}$ sur u. □

Proposition 7.2 (*automate de Büchi généralisé*)

Soit $\mathcal{G} = (Q, \Sigma, T, I, \mathcal{T})$ un automate de Büchi généralisé (défini sous sa version optimisée). Si $\exists (q, \alpha_1, q'), (q, \alpha_2, q') \in T$ avec $\alpha_2 \subsetneq \alpha_1$ et $\forall T_j \in \mathcal{T}$, $(q, \alpha_2, q') \in T_j$ implique $(q, \alpha_1, q') \in T_j$, alors on peut supprimer (q, α_2, q') de T.

Démonstration. Ce résultat est clair car si $u_i \in \alpha_2$ alors $u_i \in \alpha_1$, et ainsi toute exécution acceptante de $\mathcal{G}$ peut être considérée comme une exécution acceptante de l'automate modifié et réciproquement. □

Proposition 7.3 (*automate de Büchi*)

Soit $\mathcal{B} = (Q, \Sigma, \delta, I, R)$ un automate de Büchi (défini sous sa version optimisée). Si $\exists q \in Q\ \exists(\alpha_1, q'), (\alpha_2, q') \in \delta(q)$ tels que $\alpha_2 \subsetneq \alpha_1$, alors on peut supprimer (α_2, q') de $\delta(q)$.

Démonstration. Voir la démonstration de la proposition 7.2 □

Pour le cas particulier d'un automate de Büchi généralisé issu d'un automate alternant très faible par notre construction, la simplification peut être encore améliorée. Nous allons donc maintenant donner la construction réelle que nous avons utilisée, et prouver que cette construction est correcte comme promis.

Définition 7.6 ($\mathcal{A} \rightarrow \mathcal{G}_{\mathcal{A}}$ *optimisé*)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible.
L'automate de Büchi généralisé $\mathcal{G}_{\mathcal{A}} = (Q', \Sigma, T', I', \mathcal{T})$ est défini comme suit :
- $Q' = 2^Q$,
- $T = \{(X, \alpha, X') \mid (\alpha, X') = \bigotimes_{q \in X} \delta(q)\}$,
- T' est l'ensemble des transitions $\preccurlyeq$-minimales de T, où la relation $\preccurlyeq$ est définie comme suit : soient $t_1 = (X, \alpha_1, X_1)$, $t_2 = (X, \alpha_2, X_2)$. On a $t_1 \preccurlyeq t_2$ si $\alpha_2 \subseteq \alpha_1$, $X_1 \subseteq X_2$ et $\forall q \in Q \backslash R$, $t_2 \in T_q \Rightarrow t_1 \in T_q$,
- $I' = I$,
- $\mathcal{T} = \{T_q \mid q \in Q \backslash R\}$ où
  $T_q = \{(X, \alpha, X') \in T' \mid q \notin X' \text{ ou } \exists(\beta, X'') \in \delta(q),\ \alpha \subseteq \beta,\ X'' \subseteq X' \backslash \{q\}\}$

Théorème 7.4

Pour tout automate alternant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G}_{\mathcal{A}})$.

Démonstration.

$\boxed{\subseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$, et soit $\rho = (V, E, \sigma)$ un Q-DAG, exécution acceptante de $\mathcal{A}$ sur u. Nous allons commencer par définir par récurrence une suite d'ensembles $(V_i, E_i)_{i \geq 1}$ avec $V_i \subseteq \nu^{-1}(i)$ et $E_i \subseteq E$, afin de définir une nouvelle exécution de $\mathcal{A}$ sur u qui n'utilise que des transitions $\preccurlyeq$-minimales.

Posons $V_1 = \nu^{-1}(1)$. Supposons maintenant que V_i a été défini pour $i \geq 1$: nous allons définir V_{i+1} et $E_i \subseteq V_i \times V_{i+1}$. Par définition d'un Q-DAG, $\forall y \in V_i$, $\exists \alpha_y$ tel que $u_i \in \alpha_y$, et $(\alpha_y, X_y) \in \delta(\sigma(y))$ où $X_y = \sigma(E(y))$. Posons $\alpha = \bigcap_{y \in V_i} \alpha_y$ et $X = \bigcup_{y \in V_i} X_y$. On a clairement $u_i \in \alpha$, et $t = (\sigma(V_i), \alpha, X)$ est une transition de T. Soit $t' = (\sigma(V_i), \alpha', X')$ une transition $\preccurlyeq$-minimale de T, telle que $t' \preccurlyeq t$. Par définition, la transition t' est dans T', et $\alpha' = \bigcap_{y \in V_i} \alpha'_y$, $X' = \bigcup_{y \in V_i} X'_y$, avec $\forall y \in V_i$, $(\alpha'_y, X'_y) \in \delta(\sigma(y))$, et $u_i \in \alpha'_y$ car $\alpha \subseteq \alpha' \subseteq \alpha'_y$. De plus, $\forall y \in V_i$, si $\sigma(y) = q \in Q \backslash R$ et $t' \in T_q$, on peut choisir α'_y, X'_y tels que $q \notin X'_y$. Posons $V_{i+1} = \{y \in V \mid \lambda(y) \in X' \times \{i+1\}\}$ et $E_i = \{(y, z) \in V_i \times V_{i+1} \mid \sigma(z) \in X'_y\}$. On a bien $V_{i+1} \subseteq \nu^{-1}(i+1)$ et $E_i \subseteq E$ car $\forall y \in V_i$, $X'_y \subseteq X' \subseteq X = \sigma(E(V_i))$.

Soient $V' = \bigcup_{i \geq 1} V_i$, $E' = \bigcup_{i \geq 1} E_i$ et σ' la restriction de σ à V'. Il apparaît clairement que $\rho' = (V', E', \sigma')$ est une exécution de $\mathcal{A}$ sur u. Supposons que ρ' ne soit pas acceptante : $\mathcal{A}$ étant très faible, il existe une branche infinie de ρ' sur laquelle la suite des étiquettes est ultimement constante de limite $q \notin R$. Or si dans ρ' il existe une arête (y, z) avec $\sigma(y) = \sigma(z) = q$, alors $q \in X'_y$, donc $t' \notin T_q$. Comme $t' \preccurlyeq t$, $t \notin T_q$ et ainsi $q \in X_y$: il existe dans ρ une arête (y, z) avec $\sigma(y) = \sigma(z) = q$. Ainsi on en déduit que ρ contient une branche infinie ultimement étiquetée par q, ce qui contredit le fait que ρ est acceptante.

Nous avons maintenant une nouvelle exécution de $\mathcal{A}$, qui n'utilise que des transitions $\preccurlyeq$-minimales. Il nous reste simplement à construire une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u.

$\forall i \geq 1$, posons $X_i = \sigma(\nu^{-1}(i))$. $X_1 = \sigma(V_1) \in I$, et $\forall i \geq 0$, $\exists \alpha_i$ tel que $u_i \in \alpha_i$ et $(X_i, \alpha_i, X_{i+1}) \in T'$. Ainsi $\rho'' = X_1, X_2, \ldots$ est une exécution de $\mathcal{G}_\mathcal{A}$ sur u. Nous allons maintenant prouver que ρ'' est acceptante.

Soient $i \geq 0$ and $q \in Q \backslash R$. Nous allons prouver qu'il existe une profondeur $j \geq i$ telle que $(X_j, \alpha_j, X_{j+1}) \in T_q$.

Si $q \notin X_{i+1}$ alors $j = i$ convient. Sinon soit $j > i$ la plus petite profondeur pour laquelle il existe un nœud y avec $\lambda(y) = (q, j)$ et $\forall z \in E(y)$, $\sigma(z) \neq q$ Notons que j existe nécessairement car ρ' ne possède aucune branche infinie ultimement étiquetée par q. Par construction de ρ', $\exists(X'_y, \alpha'_y) \in \delta(q)$ avec $\alpha_j \subseteq \alpha'_y$ et $X'_y \subseteq X_{j+1} \backslash \{q\}$, d'où $(X_j, \alpha_j, X_{j+1}) \in T_q$.

Ainsi ρ'' est une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u, et donc $u \in \mathcal{L}(\mathcal{G}_\mathcal{A})$.

$\boxed{\supseteq}$ Réciproquement, soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{G}_\mathcal{A})$ et soit $\rho' = X_0, X_1, \ldots$ une exécution acceptante de $\mathcal{G}_\mathcal{A}$ sur u.

Soit $V = \{(q, i) \mid i \geq 1 \text{ et } q \in X_i\}$. V sera l'ensemble des nœuds d'un Q-DAG ρ que nous allons construire, de telle sorte que ρ soit une exécution acceptante de $\mathcal{A}$ sur u. Pour tout nœud (q, i) de V, nous définissons $\sigma((q, i)) = q$, et par construction $\nu((q, i)) = i$.

Par définition de T', $\forall y \in V$, $\exists(\alpha_y, X_y) \in \delta(\sigma(y))$ tels que, si $\nu(y) = i$, $X_y \subseteq X_{i+1}$ et $u_i \in \alpha_y$. De plus $\forall q \in Q \backslash R$, si $\exists \alpha_i \in 2^\Sigma$ tel que $u_i \in \alpha_i$ et $(X_i, \alpha_i, X_{i+1}) \in T_q$ alors si $y = (q, i) \in V$ on peut choisir α_y et X_y tels que $q \notin X_y$. Définissons l'ensemble d'arêtes $E = \{(y, z) \in V \times V \mid \nu(z) = \nu(y) + 1 \text{ et } \sigma(z) \in X_y\}$.

On voit aisément que $\rho = (V, E, \sigma)$ est une exécution de $\mathcal{A}$ sur u. Supposons que ρ ne soit pas acceptante : elle contiendrait alors une branche infinie ultimement étiquetée par $q \in Q \backslash R$. Or ρ' est acceptante, et contient une infinité de transitions dans T_q, et pour chaque transition de T_q il n'y a pas dans E d'arête dont les deux extrémités sont d'étiquette q à la profondeur correspondante.

Ainsi ρ est une exécution acceptante de $\mathcal{A}$ sur u, et donc $u \in \mathcal{L}(\mathcal{A})$. □

7.2.2 Simplification des états

Si deux états q_1 et q_2 d'un automate sont équivalents, c'est-à-dire qu'ils ont les mêmes transitions vers les mêmes états, alors ils peuvent être fusionnés, en redirigeant les transitions de destination q_2 (la nouvelle destination est q_1), et en supprimant l'état q_2 et ses transitions.

- Pour un automate alternant très faible, q_1 et q_2 sont équivalents si $\delta(q_1) = \delta(q_2)$ et $q_1 \in R \iff q_2 \in R$.
- Pour un automate de Büchi généralisé, q_1 et q_2 sont équivalents si $\forall (q', \alpha) \in Q \times 2^\Sigma$, $(q_1, \alpha, q') \in T \iff (q_2, \alpha, q') \in T$ et $\forall T_j \in \mathcal{T}$, $(q_1, \alpha, q') \in T_j \iff (q_2, \alpha, q') \in T_j$.
- Pour un automate de Büchi, q_1 et q_2 sont équivalents si $\delta(q_1) = \delta(q_2)$ et $q_1 \in R \iff q_2 \in R$.

Dans le cas particulier d'un automate de Büchi généralisé directement issu d'un automate alternant très faible par notre construction, la condition $(q_1, \alpha, q') \in T$ ne dépend que de α et de q', et donc on a simplement "q_1 et q_2 sont équivalents si $\delta(q_1) = \delta(q_2)$". Ceci devient faux une fois que des états ont été fusionnés.

Nous allons prouver que ces règles de simplification ne changent pas le langage de l'automate. Seul le cas de l'automate alternant sera traité, les autres cas pouvant être prouvés avec la même démonstration.

Proposition 7.5

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible (défini sous sa version optimisée). Si $\exists q_1, q_2 \in Q$, $q_1 \neq q_2$ tels que $\delta(q_1) = \delta(q_2)$ et $q_1 \in R \iff q_2 \in R$ alors on peut supprimer de Q l'état q_2, en remplaçant chaque occurrence de q_2 dans I et dans δ par q_1.

Démonstration. Soit $\mathcal{A}'$ l'automate modifié, après fusion des états.
Nous allons prouver que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

$\boxed{\subseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$ et soit ρ une Q-forêt, exécution acceptante de $\mathcal{A}$ sur u. En remplaçant dans ρ toutes les étiquettes q_2 par des étiquettes q_1, on obtient clairement une exécution acceptante de $\mathcal{A}'$ sur u, et $u \in \mathcal{L}(\mathcal{A}')$.

$\boxed{\supseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A}')$ et soit $\rho = (V, E, \sigma)$ une exécution acceptante de $\mathcal{A}'$ sur u. Soient $y \in V$, $(q, i) = \lambda(y)$, $X' = \sigma(E(y))$. $\exists \alpha \in 2^\Sigma$ tel que $u_i \in \alpha$ et $(\alpha, X') \in \delta'(q)$. Il existe $(\alpha, X) \in \delta(q)$ tel que le remplacement de q_2 par q_1 dans X produit X'.

- si $q_2 \notin X$, on ne change rien,
- si $q_2 \in X$ et $q_1 \notin X$, on change σ pour que les fils de x d'étiquette q_1 aient pour nouvelle étiquette q_2,
- si $q_2 \in X$ et $q_1 \in X$, on duplique un fils de x d'étiquette q_1 et son sous-arbre, et la copie de ce fils prend pour étiquette q_2.

Après avoir effectué ces opérations, on a $\sigma(E(y)) = X$.

De même, pour les racines, $\sigma(\Gamma) \in I'$ donc il existe $X \in I$ tel que le remplacement de q_2 par q_1 dans X produit $\sigma(\Gamma)$. À nouveau, trois cas sont possibles :
- si $q_2 \notin X$, on ne change rien,
- si $q_2 \in X$ et $q_1 \notin X$, on change σ pour que les racines d'étiquette q_1 aient pour nouvelle étiquette q_2,
- si $q_2 \in X$ et $q_1 \in X$, on duplique une racine d'étiquette q_1 et son sous-arbre, et la copie de cette racine prend pour étiquette q_2.

On se convaincra aisément que la Q-forêt ainsi obtenue devient une exécution acceptante de $\mathcal{A}$ sur u, et $u \in \mathcal{L}(\mathcal{A})$. □

7.2.3 Utilisation des composantes fortement connexes

Les composantes fortement connexes d'un automate peuvent aisément être calculées, en temps linéaire, à l'aide de l'algorithme de Tarjan [Tar72] par exemple. Ce calcul permet d'apporter les simplifications suivantes, sans changer le langage de l'automate :

1. Les états non accessibles peuvent être supprimés.
2. Pour un automate alternant très faible ou un automate de Büchi, l'appartenance à R d'un état qui n'est pas dans une composante fortement connexe n'a pas besoin d'être prise en compte.
3. Pour un automate alternant très faible ou un automate de Büchi, une composante qui ne contient pas d'état final et qui n'a pas de transition vers une autre composante peut être supprimée.
4. Pour un automate de Büchi généralisé, les conditions d'acceptation d'une transition entre deux composantes distinctes n'ont pas besoin d'être prises en compte.
5. Pour un automate de Büchi généralisé, une composante fortement connexe qui ne contient pas au moins une transition dans *chaque* table d'acceptation et n'a pas de transition vers une autre composante peut être supprimée.

Le fait de déclarer les conditions d'acceptation d'un état ou d'une transition comme "n'ayant pas besoin d'être prises en compte" permet d'élargir le nombre de simplifications d'états ou de transitions possibles ultérieurement.

Proposition 7.6

Les simplifications citées ci-dessus ne changent pas le langage de l'automate

Démonstration. Une exécution d'un automate de Büchi généralisé ou d'un automate de Büchi, et, de façon similaire, la suite des étiquettes d'une branche infinie d'une exécution d'un automate alternant, est une suite infinie d'éléments de Q. Cette suite est nécessairement ultimement "coincée" au sein d'un composante fortement connexe limite.

Comme l'on s'intéresse ici uniquement aux exécutions sur des mots infinis, la satisfaction des conditions finales avant l'arrivée dans la composante limite n'influe en rien sur l'acceptation d'un mot. Cette affirmation permet de justifier les affirmations 2 et 4, qui concernent des éléments n'appartenant à aucune composante : des transitions entre deux composantes, et des états qui n'appartiennent à aucun cycle.

Si une composante est “sans issue”, c'est-à-dire qu'elle n'a aucune transition vers une autre composante, alors toute exécution tombant dans cette composante n'en sortira plus. Si cette composante ne contient pas les éléments permettant de trouver un cycle acceptant (un état répété ou une transition de chaque table), alors aucune exécution qui visite cette composante ne peut être acceptante, et donc on peut supprimer cette composante sans modifier le langage accepté par l'automate. C'est ce qui permet de justifier les affirmations 3 et 5.

Quant à la première affirmation, elle est usuelle et ne pose pas de difficulté. □

7.2.4 Simplification à la volée

La *simplification à la volée* est la force principale de notre algorithme : au cours de la construction, aussitôt qu'une transition est créée, on la compare aux autres transitions issues du même état déjà créées, et on simplifie les transitions redondantes le cas échéant. Puis lorsque toutes les transitions d'un état ont été créées, on compare aussitôt cet état aux autres états créés, et on effectue une fusion si c'est possible. Ensuite une fois l'automate entièrement créé, on peut alterner les simplifications comme indiqué plus haut.

Ainsi, dans la plupart des cas, l'espace mémoire dont a besoin l'algorithme est comparable à la taille de l'automate après simplification, alors que, sans simplification à la volée, la mémoire nécessaire à la représentation de l'automate avant sa simplification peut être considérablement plus grande, même si l'automate se simplifie bien par la suite.

De plus, les calculs de simplification sont plus rapides s'ils sont effectués à la volée, puisqu'ils permettent de restreindre le nombre de transitions ou d'états auxquels une transition ou un état doit être comparé.

Prenons un exemple : sur la formule $\neg((\mathsf{G}\,\mathsf{F}\,p_1 \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\,p_n) \rightarrow \mathsf{G}(r \rightarrow \mathsf{F}\,g))$, l'automate de Büchi généralisé non simplifié possède 2^{n+1} états, alors qu'une fois simplifié on peut le réduire à 2 états. Avec la simplification à la volée, l'automate ne possède jamais plus de 3 états (dès qu'un troisième état est créé, il est fusionné avec l'un des deux autres états).

7.3 Résultats expérimentaux

Après toutes ces promesses, il reste à prouver que notre algorithme est aussi efficace qu'annoncé. C'est ce que nous allons faire ici, en exposant les comparaisons que nous avons pu faire entre notre implémentation de cet algorithme et les autres outils effectuant le même travail et disponibles à ce moment là (printemps 2001).

Ces outils transforment tous une formule LTL en un automate de Büchi classique, donc nous avons inclus la dernière étape de la construction dans ces comparaisons. Voici les outils que nous avons pu tester :

SPIN est un model-checker développé par les Bell Labs depuis 1980. Il contient en particulier un algorithme de transformation de LTL en automates de Büchi, présenté dans [GPVW95]. Le programme est écrit en C, et nous avons utilisé la version 3.4.1.

WRING est l'implémentation d'un algorithme présenté dans [SB00]. Le programme est écrit en Perl, donc la comparaison avec notre outil n'est pas forcément à prendre à la lettre, et la mémoire utilisée par l'outil lors des calculs a été relevée en utilisant la commande Unix '`top`'.

EQLTL est l'implémentation d'un algorithme présenté dans [EH00]. Ce programme n'est pas disponible au public, mais une démonstration est proposée via une interface web. La seule mesure que nous avons pu effectuer était le temps de réponse du serveur à nos requêtes, en ignorant le type de machine effectuant le calcul et sa charge. Par conséquent les chiffres que nous donnons doivent être lus avec précaution.

LTL2BA est l'implémentation que nous avons réalisée à partir de l'algorithme explicité dans cette partie. Elle est écrite en C comme SPIN, afin de pouvoir être utilisée directement pour le model-checking en utilisant une partie du code de SPIN, et afin que la comparaison entre ces deux outils soit vraiment fiable.

LTL2BA– est la même implémentation sans simplification à la volée, afin de pouvoir mesurer l'incidence de cette dernière dans les performances de notre programme.

Tous nos tests ont été effectués sur une station SUN ULTRA 10, possédant une mémoire vive de 1 Go.

Conditions d'équité La première comparaison que nous avons effectuée concerne une formule de forme très usuelle en model-checking avec un nombre paramétré de conditions d'équité :

$$\theta_n = \neg((\mathsf{G}\,\mathsf{F}\,p_1 \wedge \ldots \wedge \mathsf{G}\,\mathsf{F}\,p_n) \rightarrow \mathsf{G}(r \rightarrow \mathsf{F}\,g))$$

Les résultats sont exposés dans les tableaux 7.1 et 7.2

Opérateurs U imbriqués à droite Une autre formule de forme usuelle, et connue pour l'explosion combinatoire qu'elle génère, est l'imbriquation paramétrée d'opérateurs U (*until*) à droite :

$$\varphi_n = \neg(p_1 \,\mathsf{U}\, (p_2 \,\mathsf{U}\, (\ldots \mathsf{U}\, p_n) \ldots)$$

Les résultats sont exposés dans le tableau 7.3.

	SPIN	WRING	EQLTL	LTL2BA−	LTL2BA
θ_1	0,18	0,56	16	0,01	0,01
θ_2	4,6	2,6	16	0,01	0,01
θ_3	170	16	18	0,01	0,01
θ_4	9 600	110	25	0,07	0,06
θ_5		1 000	135	0,70	0,37
θ_6		8 400	*N/A*	12	4.0
θ_7		72 000†		220	32
θ_8				4 200	360
θ_9				97 000	3 000
θ_{10}					36 000

TAB. 7.1 – Comparaison sur les formules θ_n : temps de calcul en secondes. (*N/A*) : pas de réponse du serveur en 24 h. (†) : arrêt prématuré du programme.

	SPIN	WRING	LTL2BA−	LTL2BA
θ_1	460	4 100	9	9
θ_2	4 200	4 100	19	11
θ_3	52 000	4 200	86	19
θ_4	970 000	4 700	336	38
θ_5		6 500	1 600	48
θ_6		13 000	8 300	88
θ_7		43 000†	44 000	175
θ_8			260 000	250
θ_9			1 600 000	490
θ_{10}				970

TAB. 7.2 – Comparaison sur les formules θ_n : mémoire utilisée en ko.

	SPIN		WRING		EQLTL	LTL2BA	
	temps	mémoire	temps	mémoire	temps	temps	mémoire
φ_2	0,01	8	0,07	4 100	8	0,01	3,2
φ_3	0,03	110	0,29	4 100	8	0,01	5,5
φ_4	0,75	1 700	1,34	4 200	9	0,01	11
φ_5	43	51 000	10	4 200	11	0,01	13
φ_6	1 200	920 000	92	4 500	15	0,15	25
φ_7			720	6 000	27	9,2	48
φ_8					92	1 200	93

TAB. 7.3 – Comparaison sur les formules φ_n : temps de calcul en secondes, mémoire utilisée en ko.

	SPIN		LTL2BA	
	moyenne	maximum	moyenne	maximum
temps de calcul en secondes	14,23	4521,65	0,01	0,04
nombre d'états	5,74	56	4,51	39
nombre de transitions	14,73	223	9,67	112

Tab. 7.4 – Comparaison sur des formules aléatoires de taille fixe.

Formules aléatoires Enfin, nous avons effectué des comparaisons sur des formules générées aléatoirement. Nous avons pour cela utilisé l'outil présenté par H. Tauriainen dans [Tau99], qui génère des formules aléatoires, et compare les résultats obtenus sur ces formules et leur négation par plusieurs outils : le temps de calcul, le nombre d'états, et le nombre de transitions de l'automate de Büchi généré.

Nous avons comparé ici notre outil à SPIN uniquement, puisque les autres outils utilisent des formats de représentation des formules LTL et des automates de Büchi qui sont incompatibles avec le programme de H. Tauriainen. Les tests ont été effectués sur 200 formules aléatoires de taille fixe égale à 10, les deux outils étant évalués sur les mêmes formules. Nous avons reporté les données mesurées en moyenne et dans le pire des cas.

Le tableau 7.4 expose les résultats obtenus.

7.4 L'outil LTL2BA

Nous allons dans cette section parler plus en détail de l'outil LTL2BA que nous avons conçu.

7.4.1 Utilisation

LTL2BA est un programme développé dans le langage C. Il prend en entrée, sur la ligne de commande ou dans un fichier, une formule LTL et des commutateurs optionnels, et produit sur la sortie standard les éléments souhaités. Par défaut, il génère un automate de Büchi qui reconnaît le langage de la formule, décrit en langage Promela, qui est le langage qu'utilise le logiciel SPIN. Ceci permet d'utiliser l'outil LTL2BA en lieu et place de SPIN pour générer cet automate, puis d'utiliser SPIN pour effectuer le model-checking lui-même, car ce logiciel accepte qu'on lui fournisse directement un programme Promela au lieu de la formule LTL.

La syntaxe d'utilisation du programme est la suivante :

```
$ ltl2ba [-a] [-c] [-d] [-l] [-o] [-p] [-s] -f formula
$ ltl2ba [-a] [-c] [-d] [-l] [-o] [-p] [-s] -F file
```

Voici les options disponibles et leurs effets :

`-f formula` permet de spécifier sur la ligne de commande une formule LTL écrite en caractères ASCII, entre guillemets,

`-F file` permet de spécifier dans un fichier une formule LTL écrite en caractères ASCII,

`-a` remplace la condition $q \in X'$ par $q \in X$ dans la définition des tables T_q de l'automate de Büchi généralisé,

`-c` annule la simplification par composantes fortement connexes,

`-d` affiche tous les automates obtenus au cours du calcul : automate alternant, automate de Büchi généralisé et automate de Büchi, avec pour chacun l'automate obtenu avant la simplification et l'automate obtenu après la procédure de simplification,

`-l` annule la simplification de la formule LTL,

`-o` annule la simplification à la volée sur tous les automates générés,

`-p` annule la simplification *a posteriori* des automates, mais n'annule pas la simplification à la volée,

`-s` affiche le temps de calcul utilisé à chaque étape et la taille de chacun des automates obtenus.

La formule LTL, fournie en ligne de commande ou dans un fichier, suit la syntaxe de SPIN : les prédicats sont notés par une suite de caractères alphanumériques commençant pas une lettre minuscule, les parenthèses sont autorisées, et les opérateurs de logique sont notés de la façon suivante :

$\top$	`true`	$\neg$	`!`	$\rightarrow$	`->`	X	`X`	U	`U`
$\bot$	`false`	$\wedge$	`&&`	$\leftrightarrow$	`<->`	F	`<>`	R	`V`
		$\vee$	`||`			G	`[]`		

TAB. 7.5 – Syntaxe LTL pour LTL2BA

7.4.2 Structure du programme

Le programme LTL2BA a été développé à partir du programme libre SPIN. En effet cela facilite et rend beaucoup plus fiable la comparaison entre les deux programmes, pour les raisons suivantes :

- les deux outils sont développés dans le même langage,
- les fonctionnalités communes aux deux outils peuvent être partagées, ce qui permet d'une part de ne pas réécrire inutilement du code, et d'autre part de centrer les comparaisons de performances sur les modules qui diffèrent et non sur deux implémentations différentes d'un même module,

– il est aisé d'utiliser les mêmes formalismes et langages en entrée et en sortie du programme, ce qui simplifie les tests.

Ainsi nous avons conservé les modules d'analyse syntaxique et de simplification de la formule, et le module de gestion de la mémoire. Puis nous avons amélioré la simplification des formules, et nous avons ajouté trois modules correspondant à la génération puis la simplification de chacun des automates. Un dernier module gère enfin les structures de données et leur manipulation.

Au total, l'application LTL2BA possède 4100 lignes de code. Elle est disponible sous licence GPL et peut être téléchargée sur Internet.

7.4.3 Interface web

Une interface web a été développée pour l'utilisation de notre programme. C'est un script qui utilise l'outil LTL2BA et met en forme les résultats pour les afficher. Un copie d'écran de cette interface est exposée sur la figure 7.1.

Cette interface a pour composant principal un formulaire, contenant une zone de saisie pour la formule LTL, et une série de cases à cocher permettant de sélectionner différentes options. Elle propose, en plus des options du programme LTL2BA, les possibilités suivantes :

– le choix de l'utilisation de SPIN au lieu de LTL2BA,
– le choix entre la syntaxe de SPIN et une autre syntaxe pour LTL,
– l'affichage sous forme d'image de l'automate de Büchi généralisé et/ou de l'automate de Büchi,

Les images des automates sont générées à la volée par le script, qui fait appel au programme DOT, qui permet de dessiner de nombreux types de graphes à partir d'une simple description textuelle des nœuds et des arêtes à représenter.

Une autre interface a été développée pour LTL2BA par Michael Baldamus, elle est écrite en Java.

7.4.4 Références web

Téléchargement libre du programme LTL2BA.
`http ://liafa.jussieu.fr/∼oddoux/ltl2ba/download.html`

Interface web du programme LTL2BA.
`http ://liafa.jussieu.fr/∼oddoux/ltl2ba/`

Interface Java du programme LTL2BA.
`http ://user.it.uu.se/ michaelb/ltl2baif/`

Outil DOT de dessin de graphes.
`http ://www.research.att.com/sw/tools/graphviz/`

LTL 2 BA : fast algorithm from LTL to Büchi automata

LTL formula: GFp [Convert]

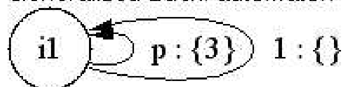
- [ ] Use Spin syntax
- [ ] Use Spin 3.4.1

- [x] Display an image of the generalised Büchi automaton
- [x] Display an image of the Büchi automaton
- [ ] Print a never claim for Spin

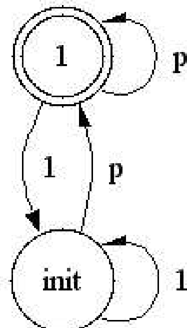
- [ ] Use verbose mode
- [ ] Display time and size statistics

- [x] Enable simplification of the formula
- [x] Enable on-the-fly automata simplification
- [x] Enable a posteriori automata simplification
- [x] Enable strongly connected components simplification
- [x] Consider second set in fj->fj (internal use)

Generalized Büchi automaton



Büchi automaton



Back to home page
Download LTL2BA
LTL2BAIF : a Java interface for LTL2BA
Denis Oddoux

FIG. 7.1 – Interface web pour LTL2BA

Troisième partie

L'extension à PLTL

Introduction

Motivations

Les logiques temporelles [Pnu77, Eme90, MP92] font partie des langages de spécification les plus utilisés, les plus populaires étant les logiques temporelles arborescentes (CTL, CTL*), et la logique temporelle linéaire (LTL). Ces trois logiques sont basées sur des opérateurs de type pur futur, qui ne tiennent pas compte de ce qui s'est passé avant l'instant présent.

Il est toutefois naturel de vouloir dans certains cas exprimer des propriétés faisant référence au passé, comme dans la formule $\mathsf{G}(g \rightarrow \mathsf{Y}(\neg g \,\mathsf{S}\, r))$, qui exprime qu'au plus une seule réponse suit une requête.

L'ajout d'opérateurs passés à LTL, formant la logique PLTL, n'augmente pas son pouvoir d'expression : à toute formule PLTL il existe une formule LTL équivalente (de même langage) [Kam68, GPSS80, LS95]. Cependant PLTL est exponentiellement plus succincte que LTL : une spécification exprimée en PLTL peut être de taille exponentiellement plus petite que la même spécification exprimée en LTL [LMS02]. Ce dernier résultat n'est pas uniquement théorique : l'ajout d'opérateurs passés permet d'écrire des spécifications plus naturelles et plus simples [LPZ85].

Le problème de l'ajout d'opérateurs passés est que cette simplification possède un coût : les problèmes de satisfaisabilité ou de model-checking deviennent plus complexes à résoudre pour PLTL que pour LTL. Cette complexité n'est pas plus grande au point de vue théorique, puisque la satisfaisabilité ou le model-checking sont PSPACE-complets pour PLTL comme pour LTL [SC85], mais au point de vue pratique.

Plusieurs algorithmes pour générer un automate de Büchi à partir d'une formule PLTL ont déjà été proposés [RMD+92, KMMP93, VW94], mais peu ont été implémentés et utilisés dans un outil de model-checking. Si cela peut paraître surprenant lorsqu'on sait que l'ajout d'opérateurs passés permet d'écrire des spécifications plus simples, plus naturelles, et plus succinctes, on peut l'expliquer par la difficulté de trouver une construction efficace. C'est ce que nous avons cherché à faire ici : trouver un algorithme qui nous permette de développer une implémentation efficace produisant un automate de Büchi reconnaissant les modèles d'une spécification exprimée en PLTL.

Choix d'une approche

Comme pour LTL, l'approche la plus simple et la plus communément utilisée est la méthode des tableaux déclaratifs [KMMP93, LPZ85, SC85]. En effet, le principe détaillé dans la section 3.3.1 s'adapte de façon simple pour traiter le cas des opérateurs passés. Si les preuves de correction et de complexité sont simples, l'implémentation de cette méthode est toutefois fortement inefficace, pour les mêmes raisons que dans le cas de LTL : l'algorithme commence, dans tous les cas, par construire l'ensemble des atomes de la formule, qui sont en nombre exponentiel en la taille de la formule.

La méthode des tableaux déclaratifs étant par conséquent impossible à implémenter, les implémentations utilisent une adaptation de la méthode des tableaux incrémentaux de la section 3.3.2 [KMMP93]. Cependant, cette adaptation n'est pas aisée, et nécessite en particulier des retours en arrière au cours de l'exploration du graphe des états, alors qu'une simple exploration en avant suffisait pour LTL. L'implémentation est donc bien plus complexe que celle, exposée dans [GPVW95], qui pouvait être utilisée pour LTL.

Revenons à LTL : la partie précédente s'est attachée à démontrer, avec succès, qu'il existe une alternative bien plus efficace que la méthode des tableaux déclaratifs pour générer un automate de Büchi à partir d'une formule LTL. Cette alternative utilise les automates alternants comme étape intermédiaire, et son implémentation s'avère extraordinairement plus rapide que les implémentations basées sur la méthode des tableaux.

Il semble alors très naturel de chercher à adapter cette méthode au cas de PLTL, et c'est ce que nous allons faire par la suite. La présence d'opérateurs passés nécessite que l'automate alternant puisse lire ses données d'entrée en avant comme en arrière, ce qui nous a naturellement amenés à utiliser des automates alternants *à double sens*.

Les automates alternants à double sens ont déjà été utilisés par le passé, et permettent de reconnaître des langages bien plus expressifs que ceux de PLTL [Var98, KPV01]. En conséquence la transformation d'un automate alternant à double sens en un automate de Büchi s'avère coûteuse, et similairement à l'utilisation des automates alternants très faibles dans le cas de LTL, nous avons été amenés à nous restreindre à une sous-classe des automates alternants à double sens, et à développer une nouvelle construction permettant de générer de façon plus efficace un automate de Büchi à partir d'un automate de cette sous-classe. Cette sous-classe, comme nous le verrons par la suite, sont les automates progressants très faibles.

Approche utilisée

Voici donc l'approche que nous utilisons : à partir d'une formule PLTL φ de taille temporelle $|\varphi|$, nous commençons par générer un automate progressant très faible possédant au maximum $|\varphi| + 1$ états, et reconnaissant le langage de φ.

Ensuite, à partir de cet automate, nous générons un automate de Büchi à mémoire, qui possédera au maximum $2^{|\varphi|+1}$ états et $|\varphi|$ conditions d'acceptation. Ce nouveau type d'automate, que nous décrirons ultérieurement, lit ses données d'entrées dans un seul sens, mais conserve toujours en mémoire l'état qu'il a visité avant l'état courant. Un tel automate avec n états et r conditions d'acceptation peut être transformé en un automate de Büchi avec au maximum $n^2 \times (r+1)$ états, ce qui nous permet en définitive d'obtenir à partir de φ un automate de Büchi généralisé avec au maximum $2^{\mathcal{O}(|\varphi|)}$ états.

A titre de comparaison, l'algorithme exposé dans [KPV01] produirait à partir de φ un automate de Büchi avec au maximum $2^{O(|\varphi|^2)}$ états, mais le langage de spécification qu'il utilise est bien plus expressif que PLTL.

Plan de la partie

Cette partie s'organise comme suit :

Dans le chapitre 8, nous définirons les automates alternants à double sens et leurs propriétés, ainsi que la méthode pour obtenir un automate alternant à double sens à partir d'une formule PLTL.

Dans le chapitre 9, nous définirons notre notion d'automates de Büchi à mémoire, et comment ils sont obtenus à partir des automates progressants très faibles.

Dans le chapitre 10, nous exposerons deux approches différentes pour effectuer le model-checking d'une formule PLTL à partir de l'automate de Büchi à mémoire obtenu.

Dans le chapitre 11, nous parlerons plus en détail de notre implémentation PLTL2BA, avec nos optimisations de structures de données, nos méthodes de simplification, et enfin les résultats expérimentaux obtenus.

L'ensemble des sujets abordés dans cette partie a fait l'objet de la rédaction et de la publication d'un article [GO03a], et d'un exposé à la conférence MFCS'03. La version longue de l'article est aussi disponible [GO03b].

Chapitre 8

Automates alternants à double sens

8.1 Définitions

Les automates classiques lisent des mots de Σ^∞ "en avant", c'est-à-dire en lisant les lettres du mot l'une après l'autre, de gauche à droite. Dans certaines circonstances, et en particulier lorsqu'on utilise les opérateurs temporels du passé de PLTL, il est naturel de vouloir revenir en arrière dans la lecture du mot quand c'est nécessaire.

Pour cela, dans les automates à double sens, une transition indique quelle nouvelle position du mot on doit lire à l'étape suivante. Il existe en général deux possibilités : l'automate *avance* sur le mot, et lira la lettre qui suit la lettre courante lors de la prochaine transition, ou il *recule*, et lira alors la lettre précédente. Par la suite, un '+1' indiquera que l'automate doit avancer et un '−1' qu'il doit reculer.

8.1.1 Automates alternants à double sens

À partir de ce point, nous considérerons toujours des mots finis *et* infinis sur Σ : en effet, nous devons maintenant traiter le cas particulier du début des mots, puisque les automates peuvent reculer, et nous traiterons de façon similaire de cas des fins de mots finis. Nous allons maintenant définir les automates alternants à double sens.

Définition 8.1 (*automate alternant à double sens*)

Un *automate alternant à double sens* est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $\delta : Q \times \Sigma \to 2^{2^Q \times 2^Q}$ est la *fonction de transition*,
- $I \subseteq 2^Q$ est la *condition initiale*,
- $F \subseteq Q$ est l'ensemble des *états finaux*.
- $R \subseteq Q$ est l'ensemble des *états répétés*.

Remarque.

Nous utilisons ici immédiatement les idées introduites dans la section 7.1.2 pour optimiser notre représentation des automates. Une définition plus classique aurait utilisé $I \in \mathcal{B}^+(Q)$ et $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q \times \{-1, +1\})$.

On définit, comme pour les automates alternants, la notion d'automates alternants à double sens faibles ou très faibles.

Définition 8.2 (*automate alternant à double sens [très] faible*)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate alternant à double sens.
$\mathcal{A}$ est dit *faible* s'il existe un partition $Q_1, \ldots, Q_n$ de Q et un ordre partiel $\preccurlyeq$ sur cette partition tels que :

- $\forall i, j$, $\forall (q, q') \in Q_i \times Q_j$, si $\exists a \in \Sigma$ tel que q' apparaît dans $\delta(q, a)$, alors $j \preccurlyeq i$,
- $\forall i$, $Q_i \subseteq R$ ou $Q_i \cap R = \emptyset$.

$\mathcal{A}$ est dit *très faible* si, de plus, tous les éléments de la partition ont pour cardinal 1. Autrement dit, il existe un ordre partiel $\preccurlyeq$ sur Q tel que $\forall q, q' \in Q$, si $\exists a \in \Sigma$ tel que q' apparaît dans $\delta(q, a)$, alors $q' \preccurlyeq q$.

Pour définir les exécutions d'un automate alternant à double sens, nous avons à nouveau besoin de la notion de Q-forêt, que nous allons étendre pour prendre en compte l'aspect 'double sens'.

Définition 8.3 (*Q-forêt*)

Une Q-forêt est une forêt étiquetée (V, E, σ, ν) telle que :

- V est l'ensemble des *nœuds* (vertices),
- $E \subseteq V \times V$ est l'ensemble des *arêtes* (edges),
- $\sigma : V \rightarrow Q$ est l'*étiquette d'état* d'un nœud,
- $\nu : V \rightarrow \mathbb{N}$ est l'*étiquette de position* d'un nœud : elle indique la position de l'automate sur son mot d'entrée. Elle vérifie $\forall (x, y) \in E$, $|\nu(x) - \nu(y)| = 1$.

On utilisera les notations suivantes en plus des notations habituelles :

- Γ est l'ensemble des *racines* de la forêt,
- $\lambda = (\sigma, \nu) : V \rightarrow Q \times \mathbb{N}$ est l'*étiquette* d'un nœud,
- $\overleftarrow{E}(x) = \{y \in E(x) \mid \nu(y) - \nu(x) = -1\}$,
- $\overrightarrow{E}(x) = \{y \in E(x) \mid \nu(y) - \nu(x) = +1\} = E(x) \backslash \overleftarrow{E}(x)$,
- $\texttt{left}(x) = E^*(\overleftarrow{E}(x)) \cup \{x\}$,
- $[x, y] = \{z \in E^*(x) \mid y \in E^*(z)\}$, $]x, y[= \{z \in E^+(x) \mid y \in E^+(z)\}$, et on définit de même $]x, y]$ et $[x, y[$.

Remarque.

Par la suite, si on manipule une Q-forêt $\rho_k = (V_k, E_k, \sigma_k, \nu_k)$, on utilisera de la même façon les notations $\texttt{left}_k(x)$ pour l'ensemble $E_k^*(\overleftarrow{E}_k(x)) \cup \{x\}$, et $[x, y]_k$ pour l'ensemble $\{z \in E_k^*(x) \mid y \in E_k^*(z)\}$, ainsi que $\texttt{left}'(x)$ et $[x, y]'$ si on manipule une Q-forêt ρ'.

Nous pouvons maintenant définir les exécutions d'un automate alternant à double sens.

Définition 8.4 (*exécution, langage d'un automate alternant à double sens*)

Soient $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate alternant à double sens et $u = u_1 u_2 \ldots \in \Sigma^\infty$.
Une *exécution* de l'automate $\mathcal{A}$ sur le mot u est une Q-forêt $\rho = (V, E, \sigma, \nu)$ telle que :
- Les racines satisfont la condition initiale : $\sigma(\Gamma) \in I$ et $\nu(\Gamma) = \{1\}$,
- Les fils d'un nœud satisfont la fonction de transition : $\forall x \in V$,
  - si $1 \leq \nu(x) \leq |u|$, $(\sigma(\overleftarrow{E}(x)), \sigma(\overrightarrow{E}(x))) \in \delta(\sigma(x), u_{\nu(x)})$.
  - sinon, $E(x) = \emptyset$.

Une exécution ρ est *acceptante* si, de plus,
- $\forall x \in V$, si $\nu(x) = 0$ ou $\nu(x) = |u| + 1$, alors $\sigma(x) \in F$,
- chaque branche infinie de la Q-forêt ρ possède un nombre infini de nœuds étiquetés dans R.

Le *langage* $\mathcal{L}(\mathcal{A})$ d'un automate alternant $\mathcal{A}$ est l'ensemble des mots de Σ^∞ sur lesquels il existe une exécution acceptante de $\mathcal{A}$.

Remarques.

Dans toute exécution $\rho = (V, E, \sigma, \nu)$ de $\mathcal{A}$ sur un mot u, $\forall x \in V$, $0 \leq \nu(x) \leq |u|+1$.

Une exécution sur un mot fini peut contenir des branches infinies, qui doivent respecter la condition sur R pour que cette exécution soit acceptante.

8.1.2 Automates progressants

De même qu'il est possible d'obtenir un plus petit automate de Büchi à partir d'un automate alternant très faible qu'à partir d'un automate alternant, il existe une sous-classe des automates alternants à double sens qui permet d'optimiser les transformations ultérieures : les automates *progressants* très faibles.

Définition 8.5 (*exécution progressante, bouclante*)

Une exécution $\rho = (V, E, \sigma, \nu)$ est dite *progressante* si toute branche infinie $x_0, x_1, \ldots$ satisfait la propriété suivante : $\forall N > 0$, $\exists i \geq 0$, $\nu(x_i) \geq N$. Un automate alternant à double sens dont toutes les exécutions sont progressantes est appelé *automate progressant*.

Une exécution $\rho = (V, E, \sigma, \nu)$ est dite *bouclante* s'il existe deux nœuds de même étiquette sur une même branche : $\exists x \in V$, $\exists y \in E^+(x)$, $\lambda(y) = \lambda(x)$. Dans le cas contraire, ρ est dite *sans boucle*. Un automate alternant à double sens dont toutes les exécutions sont sans boucle est appelé *automate sans boucle*.

On notera aisément qu'une exécution sans boucle sur un mot fini ne peut avoir de branche infinie, et qu'elle est nécessairement progressante. Plus généralement, on a la propriété suivante :

Proposition 8.1

Toute exécution sans boucle d'un automate alternant à double sens est progressante.

Démonstration. Montrons la contraposée de cette proposition : soit ρ une exécution non progressante d'un automate alternant à double sens. Par définition, ρ possède une branche infinie $x_0, x_1, \ldots$ telle que $\exists N > 0, \forall i \geq 0, \nu(x_i) < N$. Ainsi l'ensemble $\{\lambda(x_i) \mid i \geq 0\}$ est inclus dans l'ensemble $Q \times \{0, \ldots, N-1\}$ qui est fini. Par conséquent il existe une infinité de nœuds avec la même étiquette sur cette branche, donc ρ est bouclante. □

Il apparaît clairement que la réciproque de cette proposition est fausse. Cependant, on peut obtenir un résultat plus fort sur les automates :

Proposition 8.2

Un automate alternant à double sens est progressant si et seulement si il est sans boucle.

Démonstration.

$\boxed{\Rightarrow}$ Soit $\mathcal{A}$ un automate progressant : nous allons montrer que toute exécution de $\mathcal{A}$ est sans boucle. Supposons qu'il existe une exécution bouclante ρ_0 de $\mathcal{A}$ sur un mot u : il existe deux nœuds $x_0 \in V$ et $x_1 \in E^+(x)$ tels que $\lambda(x_1) = \lambda(x_0)$. Remplaçons le sous-arbre issu de x_1 par une copie du sous-arbre issu de x_0 : nous obtenons une nouvelle Q-forêt ρ_1, qui est toujours une exécution de $\mathcal{A}$ sur u. Soit x_2 la copie de x_1 que nous venons de créer : $\lambda(x_2) = \lambda(x_0)$, donc on peut répéter l'étape précédente en remplaçant le sous-arbre issu de x_2 par une copie du sous-arbre issu de x_0. La nouvelle Q-forêt ρ_2 ainsi obtenue est toujours une exécution de $\mathcal{A}$ sur u. Répétant cette opération à l'infini, on construit une suite $(\rho_n)_{n \geq 0}$ d'exécutions de $\mathcal{A}$ sur u. ρ_n et ρ_{n+1} étant identiques jusqu'à la profondeur de x_{n+1}, qui croît strictement avec n, la suite converge vers une Q-forêt ρ, qui est toujours une exécution de $\mathcal{A}$ sur u. De plus par construction ρ contient une branche infinie $x_0, x_1, \ldots$ sur laquelle ν est majorée. Ceci contredit l'hypothèse que $\mathcal{A}$ est progressant. Ainsi $\mathcal{A}$ est sans boucle.

$\boxed{\Leftarrow}$ Soit $\mathcal{A}$ un automate sans boucle : toute exécution de $\mathcal{A}$ est sans boucle, et donc progressante d'après la proposition 8.1. Ainsi $\mathcal{A}$ est progressant. □

Si toutes les exécutions progressantes d'un automate alternant à double sens ne sont pas sans boucle, on a toutefois un résultat qui s'en approche : si un automate alternant très faible à double sens possède une exécution progressante sur un mot, alors il possède aussi une exécution sans boucle sur le même mot. C'est ce que nous allons prouver par la suite.

Nous commençons par définir la notion d'altération d'une exécution :

Définition 8.6 (*altération d'une exécution*)

Soit $\rho = (V, E, \sigma, \nu)$ une exécution d'un automate alternant à double sens $\mathcal{A}$ sur un mot u. Une exécution $\rho' = (V', E', \sigma', \nu')$ de $\mathcal{A}$ sur u est "une altération de ρ en x" si x est un nœud commun aux deux Q-forêts, et que les deux Q-forêts sont identiques en dehors des sous-arbres issus de x. De plus, si ρ est progressante [resp. acceptante], on impose que ρ' soit progressante [resp. acceptante] aussi.

Nous allons maintenant prouver une suite de lemmes nous menant à la proposition finale.

Lemme 8.3

Soit $\rho = (V, E, \sigma, \nu)$ une exécution d'un automate alternant à double sens $\mathcal{A}$ sur un mot u. Soient $x \in V$ et $y \in E^*(x)$ tels que $\lambda(y) = \lambda(x)$. Soit ρ' la forêt obtenue en remplaçant les fils de x par les fils de y : $\rho' = (V', E', \sigma', \nu')$ où on définit $V' = (V \backslash E^+(x)) \cup E^+(y)$, $E' = E_{|V' \times V'} \cup (\{x\} \times E(y))$, $\sigma' = \sigma_{|V'}$ et $\nu' = \nu_{|V'}$. ρ' est une altération de ρ en x.

Démonstration. Puisque $\lambda(y) = \lambda(x)$, il apparaît clairement que ρ' est une exécution de $\mathcal{A}$ sur u, identique à ρ en dehors du sous-arbre issu de x. Considérons une branche infinie de ρ' : soit elle ne contient pas le nœud x et c'est aussi une branche de ρ, soit elle contient le nœud x, et c'est une branche de ρ à laquelle on a ôté le segment fini $]x, y]$. Il est alors clair que si ρ est progressante [resp. acceptante] alors ρ' est progressante [resp. acceptante]. □

Lemme 8.4

Soit $\rho = (V, E, \sigma, \nu)$ une exécution progressante d'un automate alternant très faible à double sens $\mathcal{A}$ sur un mot u. Soit $x \in V$: on peut construire une altération de ρ en x telle que dans le sous-arbre gauche issu de x, aucun nœud n'a la même étiquette que x : $\forall y \in \texttt{left}(x)$, si $\lambda(y) = \lambda(x)$, alors $y = x$.

Démonstration. Soit $y \in V$: si y possède un descendant à gauche de même étiquette, notons $\texttt{next}(y)$ le premier descendant dans ce cas : $\texttt{next}(y) \in E^*(\overleftarrow{E}(y))$, $\lambda(\texttt{next}(y)) = \lambda(y)$ et $\forall z \in]y, \texttt{next}(y)[$, $\lambda(z) \neq \lambda(y)$. Sinon, posons $\texttt{next}(y) = \perp$, et $\texttt{next}(\perp) = \perp$. $\mathcal{A}$ étant très faible, si $\texttt{next}(y) \neq \perp$, alors $\forall z \in]y, \texttt{next}(y)[$, $\sigma(z) = \sigma(y)$ et $\nu(z) < \nu(y)$.

Posons $y_0 = x$ et $\forall n \geq 0$, $y_{n+1} = \texttt{next}(y_n)$. Si $\forall n \geq 0$, $y_n \neq \perp$, alors sur la branche infinie des y_n, $\nu(x)$ est un majorant de ν, ce qui est impossible puisque ρ est progressant. Ainsi il existe un entier n tel que $y_n \neq \perp$ et $\texttt{next}(y_n) = \perp$. Puisque $\lambda(x) = \lambda(y_n)$ on peut utiliser le lemme 8.3 en x et y_n : nous obtenons ainsi une altération de ρ en x qui satisfait les conditions du lemme. □

Lemme 8.5

Soit $\rho = (V, E, \sigma, \nu)$ une exécution progressante d'un automate très faible à double sens $\mathcal{A}$ sur un mot u. Soit $x \in V$ d'étiquette (q, n) : on peut construire une altération ρ' de ρ en x telle que dans le sous-arbre gauche issu de x, il existe au maximum un nœud d'étiquette d'état q à chaque position : $\forall y, z \in \texttt{left}'(x)$, $\forall k \in \mathbb{N}$, si $\lambda(y) = \lambda(z) = (q, k)$, alors $y = z$.

Démonstration. Soit $H(l)$ la propriété suivante : il existe une altération $\rho_l = (V_l, E_l, \sigma_l, \nu_l)$ de ρ en x et $\exists y_l \in (\overleftarrow{E_l})^l(x)$ tel que $\sigma(y_l) = q$ et $\forall z \in \texttt{left}_l(x)$ $\sigma(z) = q \Rightarrow z \in [x, y_l]_l$ ou $\nu(z) < n - l$. Nous allons prouver que $H(0)$ est vraie, et que si $H(l)$ est vraie pour $l \geq 0$ alors ρ_l satisfait les conditions du lemme, ou alors $H(l+1)$ est vraie.

• Si $l = 0$, appliquons le lemme 8.4 à ρ en x : nous obtenons une altération ρ_0 de ρ en x telle que $\forall z \in \texttt{left}_0(x)$, $\lambda(z) = (q, n) \Rightarrow z = x$. Posons $y_0 = x \in (\overleftarrow{E_0})^0(x)$. Soit $z \in \texttt{left}_0(x)$ tel que $\sigma(z) = q$: si $\nu(z) > n$ alors $\exists z' \in]x, z[$ tel que $\lambda(z') = (q, n)$, ce qui est impossible. Ainsi $\nu(z) < n$, ou alors, si $\nu(z) = n$, $z = x \in [x, y_0]$. $H(0)$ est vraie.

• Supposons que $H(l)$ est vraie pour $0 \leq l \leq n$. Deux cas sont possibles :

(1) Si y_l possède un fils gauche y_{l+1} d'étiquette d'état q ($l < n$), alors supprimons tous les autres fils gauches de y_l d'étiquette d'état q et leurs sous-arbres. Appliquons le lemme 8.4 à ρ_l en y_{l+1} : nous obtenons une altération ρ_{l+1} de ρ_l en y_{l+1} (ρ_{l+1} est aussi une altération de ρ en x) telle que, excepté y_{l+1}, aucun nœud de $\texttt{left}_{l+1}(y_{l+1})$ n'est étiqueté par $(q, n - l - 1)$.

Soit $z \in \texttt{left}_{l+1}(x)$ tel que $\sigma(z) = q$. Puisque $\texttt{left}_{l+1}(x) \subseteq \texttt{left}_l(x)$, $z \in [x, y_l]_l$ ou $\nu(z) < n - l$. Dans le deuxième cas, $z \in E^*_{l+1}(y_{l+1})$. Or d'après la construction du lemme 8.4, on a $E^*_{l+1}(y_{l+1}) \subseteq \texttt{left}_l(y_l)$. Ainsi, si $z \in E^*_{l+1}(y_{l+1})$ et $\sigma(z) = q$, alors $\nu(z) < n - l$. Donc $\forall z \in \overrightarrow{E_{l+1}}(y_{l+1})$, $\sigma(z) \neq q$, et puisque $\mathcal{A}$ est très faible, $\forall z \in E^*_{l+1}(\overrightarrow{E_{l+1}}(y_{l+1}))$, $\sigma(z) \neq q$. Ainsi $\forall z \in \texttt{left}_{l+1}(x)$ tel que $\sigma(z) = q$, $z = y_{l+1}$ ou $\nu(z) < n - l - 1$. Ainsi $\exists y_{l+1} \in (\overleftarrow{E_{l+1}})^{l+1}(x)$ tel que $\sigma(y_{l+1}) = q$ et $\forall z \in \texttt{left}_{l+1}(x)$, $z \in [x, y_{l+1}]_{l+1}$ ou $\nu(z) < n - l - 1$: $H(l+1)$ est vraie.

(2) Sinon, y_l n'a pas de fils gauche d'étiquette d'état q (c'est toujours vrai pour $l = n$, ce qui assure la terminaison de l'induction). Ainsi $\forall z \in \texttt{left}_l(x)$ tel que $\sigma(z) = q$, $z \in [x, y_l]_l$. Or $y_l \in (\overleftarrow{E_l})^l(x)$ donc ν est strictement décroissante sur $[x, y_l]_l$: il existe au maximum un nœud d'étiquette q à chaque position dans le sous-arbre issu de x. ρ_l satisfait le lemme. □

Lemme 8.6

Soit $\rho = (V, E, \sigma, \nu)$ une exécution progressante d'un automate alternant très faible à double sens $\mathcal{A}$ sur un mot u. Soit $x \in V$ d'étiquette (q, n) : on peut construire une altération ρ' de ρ en x telle que dans le sous-arbre issu de x, il existe au maximum un nœud d'étiquette d'état q à chaque position : $\forall y, z \in E'^*(x)$, $\forall k \in \mathbb{N}$, si $\lambda(y) = \lambda(z) = (q, k)$, alors $y = z$.

Démonstration. Soit $H(l)$ la propriété suivante : il existe une altération ρ_l de ρ en x, $\exists \overleftarrow{y}_l \in (\overleftarrow{E_l})^*(x)$ et $\exists \overrightarrow{y}_l \in (\overrightarrow{E_l})^l(x)$ tels que $\sigma(\overleftarrow{y}_l) = \sigma(\overrightarrow{y}_l) = q$ et $\forall z \in E_l^*(x)$, si $\sigma(z) = q$ alors $z \in [x, \overleftarrow{y}_l]_l \cup [x, \overrightarrow{y}_l]_l$ ou $z \in E_l^*(\overrightarrow{y}_l)$. Nous allons prouver que $H(0)$ est vraie, et que si $H(l)$ est vraie pour $l \geq 0$ alors ρ_l satisfait les conditions du lemme, ou alors $H(l+1)$ est vraie.

• Si $l = 0$, posons $\overleftarrow{y}_0 = \overrightarrow{y}_0 = x \in (\overleftarrow{E})^0(x) = (\overrightarrow{E})^0(x) : \forall z \in E^*(x), z \in E^*(\overrightarrow{y}_0) : \rho_0 = \rho$ convient, et $H(0)$ est vraie.

• Supposons que $H(l)$ est vraie pour $l \geq 0$. Appliquons le lemme 8.5 à ρ_l en $\overrightarrow{y}_l$: nous obtenons une altération ρ_{l+1} de ρ en x telle que les descendants à gauche de $\overrightarrow{y}_l$ d'étiquette d'état q se réduisent à une suite finie $z_0, z_1, \ldots, z_m$ de nœuds ($z_0 = \overrightarrow{y}_l$) tels que $z_i \in (\overleftarrow{E_l})^i(\overrightarrow{y}_l)$. Trois cas sont possibles :

(1) Si $m = 0$ et $\overrightarrow{y}_l$ possède un fils droit $\overrightarrow{y}_{l+1}$ d'étiquette d'état q, alors supprimons tous les autres fils droits de $\overrightarrow{y}_l$ d'étiquette d'état q et leurs sous-arbres. Ainsi $\overrightarrow{y}_{l+1}$ est le seul fils de $\overrightarrow{y}_l$ d'étiquette d'état q, et en posant $\overleftarrow{y}_{l+1} = \overleftarrow{y}_l$, on a $\forall z \in E_{l+1}^*(x)$, si $\sigma(z) = q$ alors $z \in [x, \overleftarrow{y}_{l+1}]_{l+1} \cup [x, \overrightarrow{y}_{l+1}]_{l+1}$ ou $z \in E_{l+1}^*(\overrightarrow{y}_{l+1})$: $H(l+1)$ est vraie (voir figure 8.1 : les états d'étiquette q sont entourés).

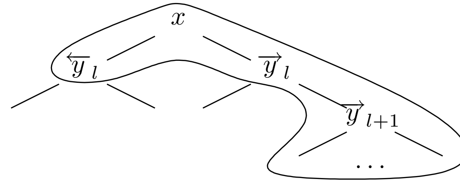


FIG. 8.1 – Cas (1)

(2) Si $m = 0$ et $\overrightarrow{y}_l$ ne possède pas de fils droit d'étiquette d'état q dans ρ_{l+1}, alors $\rho' = \rho_{l+1}$ satisfait le lemme.

(3) Si $0 < m \leq l$, alors $\lambda(z_m) = (q, l + n - m)$ et z_m n'a aucun fils d'étiquette d'état q. En appliquant le lemme 8.3 en $\overrightarrow{y}_{l-m}$ et z_m sur ρ_{l+1}, on obtient une altération ρ' de ρ en x dont on peut vérifier qu'elle satisfait le lemme : les seuls nœuds d'étiquette d'état q sont $[x, \overleftarrow{y}_l]' \cup [x, z_m]'$, et il en existe au maximum un à chaque position car $\overleftarrow{y}_l \in (\overleftarrow{E'})^*(x)$ et $z_m \in (\overrightarrow{E'})^*(x)$ (voir figure 8.2 : les états d'étiquette q sont entourés).

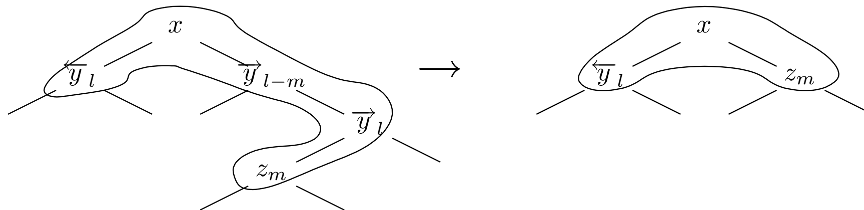


FIG. 8.2 – Cas (3)

(4) Enfin, si $m \geq l$, alors $\lambda(z_l) = (q, n)$. En appliquant le lemme 8.3 en x et z_l sur ρ_{l+1}, on obtient une altération ρ' de ρ en x dont on peut vérifier qu'elle satisfait le lemme : les seuls nœuds d'étiquette d'état q sont $[x, z_m]'$, et il en existe au maximum un à chaque position car $z_m \in (\overleftarrow{E'})^*(x)$ (voir figure 8.3 : les états d'étiquette q sont entourés).

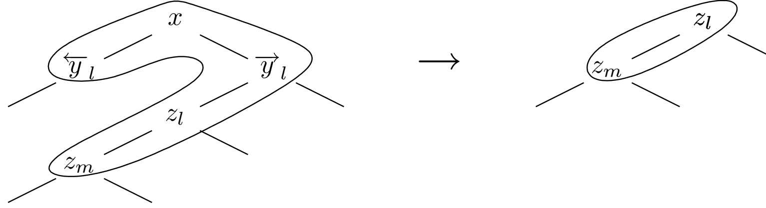


FIG. 8.3 – Cas (4)

Les cas (2), (3) et (4) stoppent la récurrence. Si seul le cas (1) est rencontré, on construit une suite infinie $(\rho_l)_{l\geq 0}$ d'altérations de ρ en x. ρ_l et ρ_{l+1} étant identiques jusqu'à la profondeur de $\overrightarrow{y}_l$, strictement croissante avec l, la suite converge vers une Q-forêt ρ', qui est toujours une exécution de $\mathcal{A}$ sur u. De plus la seule branche infinie de ρ' qui ne finisse pas comme une branche infinie de ρ est la branche des $(\overrightarrow{y}_l)_{l\geq 0}$. Cette branche est clairement progressante, donc ρ' est progressante, et si ρ est acceptante, alors le fait que le cas (1) se reproduise infiniment souvent implique que $q \in F$, par conséquent la nouvelle branche est ultimement étiquetée dans F, et ρ' est acceptante. Ainsi ρ' est une altération de ρ en x. □

Voici maintenant la proposition annoncée, que nous allons prouver à l'aide du lemme précédent.

Proposition 8.7

Soient $\mathcal{A}$ un automate alternant très faible à double sens, et ρ une exécution progressante de $\mathcal{A}$ sur un mot u. Il existe une exécution sans boucle de $\mathcal{A}$ sur u, acceptante si ρ est acceptante.

Démonstration. Soit $(q_1, \ldots, q_n)$ un ordonnancement de Q compatible avec l'ordre partiel $\preccurlyeq$ sur Q induit par le fait que $\mathcal{A}$ est très faible : $\forall 1 \leq i < j \leq n$, $q_i \succ q_j$.

Soit $H(i)$ la propriété suivante : il existe une exécution progressante de $\mathcal{A}$ sur u, acceptante si ρ est acceptante, telle que $\forall x \in V$, $\forall y \in E^*(x)$, si $\sigma(x) = \sigma(y) = q_j$ avec $j \leq i$ et $\nu(x) = \nu(y)$, alors $y = x$. $H(0)$ est clairement vraie (ρ satisfait les conditions), et nous allons prouver que si $H(i)$ est vraie pour $0 \leq i < n$ alors $H(i+1)$ est vraie.

Supposons que $H(i)$ est vraie pour $0 \leq i < n$. Appliquons le lemme 8.6 sur tous les nœuds de ρ_i d'étiquette d'état q_{i+1} dont le père a pour étiquette d'état un q_j avec $j \leq i$. Notons qu'il existe probablement une infinité de tels nœuds, mais puisque l'automate $\mathcal{A}$ est très faible, ils sont tous sur des branches différentes. Le lemme 8.6 ne modifiant que le sous-arbre du nœud auquel il est appliqué, cet algorithme définit proprement une exécution progressante ρ_{i+1} de $\mathcal{A}$ sur u, acceptante si ρ l'est.

Soient $x \in V_{i+1}$, $y \in E^*_{i+1}(x)$ tels que $\lambda_{i+1}(x) = \lambda_{i+1}(y) = (q_j, k)$ avec $j \leq i+1$. Si $j \leq i$, alors par construction x et y apparaissent dans ρ_i et n'ont pas été touchés par la construction (celle-ci n'affecte que les nœuds d'étiquette d'état $q \preccurlyeq q_{i+1}$). Ainsi $y \in E^*_i(x)$, et par hypothèse de récurrence, $x = y$. Sinon, soit z le plus vieil ancêtre de x dans ρ_{i+1} d'étiquette d'état q_{i+1} : c'est à z que le lemme 8.6 a été appliqué. $x, y \in E^*_{i+1}(z)$ et $\lambda_{i+1}(x) = \lambda_{i+1}(y)$ donc le lemme 8.6 implique que $x = y$. Ainsi ρ_{i+1} satisfait les conditions, et $H(i+1)$ est vraie.

Ainsi par récurrence la propriété $H(n)$ est vraie, et elle traduit exactement que l'exécution ρ_n est sans boucle, acceptante si ρ l'est. $\square$

8.2 Exemple

Voici un exemple d'automate alternant à double sens. Il s'agit de l'automate généré à partir de la formule $\mathsf{F}(\mathsf{Y}\,\mathsf{O}\,p \wedge \mathsf{X}(q\,\mathsf{U}\,r))$ par l'algorithme présenté ultérieurement. Il est présenté sur la figure 8.4.

Il n'est pas très aisé de représenter de façon très lisible un automate alternant à double sens. Pour mieux comprendre nos notations sur ce dessin, voici quelques explications :

- $\delta(1, \{\}) = \{(\emptyset, \{1\}), (\{2\}, \{3\})\}$,
- $\delta(2, \{\}) = \{(\{2\}, \emptyset)\}$,
- $\delta(2, \{p\}) = \{(\emptyset, \emptyset)\}$,
- $\delta(3, \{q\}) = \{(\emptyset, \{3\})\}$,
- $\delta(3, \{r\}) = \{(\emptyset, \emptyset)\}$,

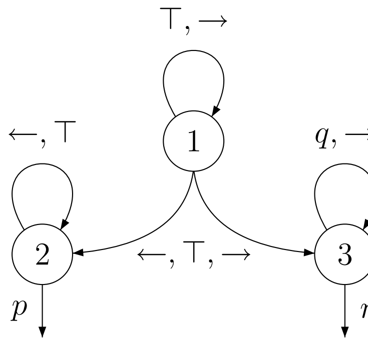


FIG. 8.4 – Automate alternant à double sens $\mathcal{A}_{\mathsf{F}(\mathsf{Y}\,\mathsf{O}\,p \wedge \mathsf{X}(q\mathsf{U}r))}$.

8.3 Des automates alternants très faibles à double sens aux automates progressants

Nous allons maintenant prouver que les automates progressants très faibles ont le même pouvoir d'expression que les automates alternants très faibles à double sens. Ce résultat est important, puisque, combiné à la construction d'un automate de Büchi généralisé à partir d'un automate progressant très faible présentée dans le chapitre 9, il permet d'obtenir un procédé efficace pour transformer un automate alternant très faible à double sens en un automate de Büchi généralisé.

Cependant, ce résultat n'est pas nécessaire pour notre construction globale de PLTL vers les automates de Büchi généralisés, puisque nous sommes capables d'obtenir directement un automate progressant très faible à partir d'une formule PLTL (voir la section 8.4).

8.3.1 Construction

L'intuition de la construction est la suivante : nous allons dupliquer l'ensemble des états en créant pour chaque état $q \in Q$ un copie $\overline{q}$. On va ensuite modifier la fonction de transition, pour que dans une exécution de $\mathcal{A}$, un nœud d'étiquette d'état q peut avoir un fils d'étiquette d'état q à gauche seulement, et un nœud d'étiquette d'état $\overline{q}$ peut avoir un fils d'étiquette d'état $\overline{q}$ à droite seulement, tout en gardant un automate très faible : sur toute branche infinie, la suite des étiquettes d'état sera ultimement stationnaire, et la limite sera nécessairement un état de la forme $\overline{q}$, impliquant que ν soit ultimement strictement croissante sur chaque branche infinie.

Définition 8.7 ($\mathcal{A} \to \mathcal{A}_p$)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate alternant très faible à double sens. L'automate alternant à double sens $\mathcal{A}_p = (Q', \Sigma, \delta', I', F', R')$ est défini comme suit :

- $Q' = Q \cup \overline{Q}$ où $\overline{Q} = \{\overline{q} \mid q \in Q\}$ est une copie de Q,
- δ' est définie comme suit, $\forall (p, a) \in Q \times \Sigma$:
  - $\delta'(p, a) = \{(X, Y_{\overline{p} \leftarrow p}) \mid (X, Y) \in \delta(p, a)\}$,
    où $Y_{q \leftarrow p} = Y$ si $p \notin Y$ et $Y_{q \leftarrow p} = Y \backslash \{p\} \cup \{q\}$ sinon,
  - $\delta'(\overline{p}, a) = \{(X, Y_{\overline{p} \leftarrow p}) \mid (X, Y) \in \delta(p, a) \wedge p \notin X\}$ si $p \notin R$, et
    $\delta'(\overline{p}, a) = \{(X \backslash \{p\}, Y_{\overline{p} \leftarrow p}) \mid (X, Y) \in \delta(p, a)\}$ si $p \in R$,
- $I' = I$,
- $F' = F \cup \overline{F}$,
- $R' = \overline{R}$.

Nous commençons par prouver que l'automate $\mathcal{A}_p$ est progressant et très faible.

Proposition 8.8

Pour tout automate alternant très faible à double sens $\mathcal{A}$, l'automate $\mathcal{A}_p$ est un automate progressant très faible.

Démonstration.

- L'automate $\mathcal{A}$ étant très faible, il définit un ordre partiel $\leq$ sur Q. Considérons la relation $\prec$ induite sur Q' par les relations suivantes :
  - $\forall p \in Q$, $\overline{p} \prec p$,
  - $\forall p < q \in Q$, $p \prec \overline{q}$.

  La clôture réflexive et transitive $\preccurlyeq$ de la relation $\prec$ est un ordre partiel sur Q'. Il est aisé de vérifier avec cet ordre partiel que $\mathcal{A}_p$ est très faible.
- Montrons maintenant que $\mathcal{A}_p$ est progressant : supposons qu'il existe une exécution non progressante $\rho = (V, E, \sigma, \nu)$ de $\mathcal{A}_p$ sur un mot u. D'après la proposition 8.1, ρ est bouclante, donc il existe $x \in V$ et $y \in E^+(x)$ tels que $\lambda(x) = \lambda(y)$. L'automate $\mathcal{A}_p$ étant très faible, il existe $x' \in V$ et $y' \in E(E(x'))$ tels que $\lambda(x') = \lambda(y')$: ainsi $y' \in \overleftarrow{E}(\overrightarrow{E}(x'))$ ou $y' \in \overrightarrow{E}(\overleftarrow{E}(x'))$. Or la définition de δ' implique qu'aucun nœud de ρ d'étiquette d'état dans Q [resp. $\overline{Q}$] ne peut avoir de fils droit [resp. gauche] avec la même étiquette d'état. Ainsi $\sigma(y') \prec \sigma(x')$, ce qui contredit nos hypothèses. Toute exécution de $\mathcal{A}_p$ est progressante, donc $\mathcal{A}_p$ est un automate progressant. □

8.3.2 Preuve de l'équivalence

Nous allons maintenant prouver que $\mathcal{A}$ et $\mathcal{A}_p$ ont le même langage, en prouvant au préalable un lemme intermédiaire.

Lemme 8.9

De toute exécution acceptante d'un automate alternant très faible à double sens $\mathcal{A}$ on peut construire une autre exécution acceptante de $\mathcal{A}$ sur le même mot, telle qu'il n'existe aucune paire (x, y), avec $y \in E^+(x)$, $\lambda(y) = \lambda(x)$ et $\sigma(x) \notin R$.

Démonstration. Soit $\rho_0 = (V_0, E_0, \sigma, \nu)$ une exécution acceptante de $\mathcal{A}$ sur un mot u. Nous allons construire par induction une suite $(\rho_n)_{n\geq 0}$ d'exécutions acceptantes de $\mathcal{A}$ sur u de la façon suivante :

Soient $n \geq 0$, et $\rho_n = (V_n, E_n, \sigma, \nu)$ une exécution acceptante de $\mathcal{A}$ on u. Pour tout nœud x à la profondeur n dans ρ_n, si $\sigma(x) \notin R$ et $\exists y \in E_n^+(x)$ tel que $\lambda(y) = \lambda(x)$, on peut trouver $x' \in E_n^+(x)$ tel que $\lambda(x') = \lambda(x)$ et $\forall y \in E_n^+(x')$, $\lambda(y) \neq \lambda(x)$. x' existe nécessairement, sinon ρ_n contiendrait une branche infinie ultimement étiquetée par $\sigma(x) \notin R$. Remplaçons les fils de x par les fils de x' : nous obtenons ainsi une altération ρ_{n+1} de ρ_n en x, exécution acceptante de $\mathcal{A}$ sur u.

Les exécutions ρ_n et ρ_{n+1} étant identiques jusqu'à la profondeur n, la suite $(\rho_n)_{n\geq 0}$ converge vers une Q-forêt $\rho = (V, E, \sigma, \nu)$, qui est toujours une exécution de $\mathcal{A}$ sur u, et dans laquelle il n'existe pas de paire (x, y) avec $y \in E^+(x)$, $\lambda(y) = \lambda(x)$, et $\sigma(x) \notin R$.

Prouvons maintenant que ρ est acceptante : par construction, tout nœud x de ρ possède la même étiquette qu'un nœud de ρ_0, par conséquent, puisque ρ_0 est acceptante, $\nu(x) \in \{0, |u| + 1\} \Rightarrow \sigma(x) \in F$. Soit $x_0, x_1, \ldots$ une branche infinie de ρ. Par construction,

$\forall i \geq 0$, $x_i \in V_0$ et $x_{i+1} \in E_0^+(x_i)$ (en effet chaque exécution ρ_{n+1} est construite à partir de ρ_n en branchant certains nœuds à un ancêtre de leur père). Par conséquent il existe une branche infinie de ρ_0 qui traverse les nœuds $x_0, x_1, \ldots$ $\mathcal{A}$ étant très faible et ρ_0 acceptante, la suite $(\sigma(x_i))_{i\geq 0}$ est ultimement constante et sa limite est dans R : ρ est acceptante. □

Théorème 8.10

Pour tout automate alternant très faible à double sens $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_p)$.

Démonstration.

$\boxed{\subseteq}$ Soient $u \in \mathcal{L}(\mathcal{A})$ et $\rho = (V, E, \sigma, \nu)$ une exécution acceptante de $\mathcal{A}$ sur u. En utilisant le lemme 8.9, on peut supposer qu'il n'existe par dans ρ de paire (x, y) avec $y \in E^+(x)$, $\lambda(y) = \lambda(x)$ et $\sigma(x) \notin R$. Nous allons maintenant construire une exécution acceptante $\rho' = (V', E', \sigma', \nu')$ de $\mathcal{A}_p$ sur u.

Commençons par définir σ' sur V par $\sigma'(y) = \overline{\sigma(y)}$ si $\exists x \in V$ tel que $y \in \overrightarrow{E}(x)$ et $\sigma(x) = \sigma(y)$ (y est le fils droit d'un nœud de même étiquette d'état), et $\sigma'(y) = \sigma(y)$ sinon. Ensuite définissons $V' \subseteq V$ selon le procédé suivant : $\forall x \in V$, $\forall y \in \overleftarrow{E}(x)$ tels que $\sigma'(x) = \overline{\sigma(y)}$, ôtons de V tous les nœuds de $E^*(y)$. Enfin soient E' et ν' les restrictions de E et ν à V'.

Prouvons tout d'abord que ρ' est une exécution de $\mathcal{A}_p$ sur u : aucune racine n'a été supprimée de V pour obtenir V', donc $\Gamma = \Gamma'$, $\sigma'(\Gamma') = \sigma(\Gamma) \in I = I'$, et $\nu'(\Gamma') = \nu(\Gamma) = \{1\}$. Soit $x \in V'$: si $\nu'(x) \in \{0, |u| + 1\}$, alors $E'(x) = E(x) = \emptyset$. Sinon, nous allons prouver que $(\sigma'(\overleftarrow{E'}(x)), \sigma'(\overrightarrow{E'}(x))) \in \delta'(\sigma'(x), u_{\nu'(x)})$:

- si $\sigma'(x) = p \in Q$, par construction de ρ', $\sigma'(x) = \sigma(x)$, $\sigma'(\overleftarrow{E'}(x)) = \sigma(\overleftarrow{E}(x))$ et $\sigma'(\overrightarrow{E'}(x)) = \sigma(\overrightarrow{E}(x))_{\overline{p}\leftarrow p}$. Comme on a $(\sigma(\overleftarrow{E}(x)), \sigma(\overrightarrow{E}(x))) \in \delta(\sigma(x), u_{\nu(x)})$, on en déduit le résultat attendu.
- si $\sigma'(x) = \overline{p} \in \overline{Q}$, et $p \notin \sigma(\overleftarrow{E}(x))$, par construction de ρ', $\sigma'(x) = \overline{\sigma(x)}$, $\sigma'(\overleftarrow{E'}(x)) = \sigma(\overleftarrow{E}(x))$ et $\sigma'(\overrightarrow{E'}(x)) = \sigma(\overrightarrow{E}(x))_{\overline{p}\leftarrow p}$. De même, on en déduit le résultat attendu.
- si $\sigma'(x) = \overline{p} \in \overline{Q}$ et $\exists y \in \overleftarrow{E}(x)$ avec $\sigma(y) = p$, par construction de ρ' et d'après nos hypothèses sur ρ, on a $p \in R$, $\sigma'(x) = \overline{\sigma(x)}$, $\sigma'(\overleftarrow{E'}(x)) = \sigma(\overleftarrow{E}(x)) \backslash \{p\}$ et $\sigma'(\overrightarrow{E'}(x)) = \sigma(\overrightarrow{E}(x))_{\overline{p}\leftarrow p}$. On en déduit encore le résultat attendu.

Montrons que ρ' est acceptante : d'après la construction de ρ', $\forall x \in V'$, $\sigma'(x) = \sigma(x)$ ou $\sigma'(x) = \overline{\sigma(x)}$ et $\nu'(x) = \nu(x)$. Ainsi $\nu'(x) \in \{0, |u|+1|\} \Rightarrow \sigma'(x) \in F' = F \cup \overline{F}$. Considérons une branche infinie de ρ' : $\mathcal{A}_p$ étant très faible, cette branche est ultimement étiquetée par un état $q \in Q \cup \overline{Q}$. $\mathcal{A}_p$ étant progressant, ν' est strictement croissant sur cette branche à partir d'une certaine profondeur. Par conséquent, par définition de δ', on a nécessairement $q = \overline{p} \in \overline{Q}$. De plus, tous les nœuds de cette branche apparaissent dans le même ordre que dans une branche infinie de ρ. ρ étant acceptante, nécessairement $p \in R$ donc $q \in R' = \overline{R}$: ρ' est une exécution acceptante de $\mathcal{A}_p$ sur u, donc $u \in \mathcal{L}(\mathcal{A}_p)$.

$\boxed{\supseteq}$ Soient $u \in \mathcal{L}(\mathcal{A}_p)$ et $\rho' = (V', E', \sigma', \nu)$ une exécution acceptante de $\mathcal{A}_p$ sur u. Pour tout nœud $x \in V'$ tel que $\sigma'(x) = p$ ou $\sigma'(x) = \overline{p}$ avec $p \in Q$, posons $\sigma_0(x) = p$. Soit $x \in V'$ tel que $1 \leq \nu(x) \leq |u|$, et supposons que $(\sigma_0(\overleftarrow{E'}(x)), \sigma_0(\overrightarrow{E'}(x))) \notin \delta(\sigma_0(x), u_{\nu(x)})$. Nécessairement $\sigma'(x) = \overline{p}$ avec $p \in R$ et $(\sigma_0(\overleftarrow{E'}(x)) \cup \{p\}, \sigma_0(\overrightarrow{E'}(x))) \in \delta(\sigma_0(x), u_{\nu(x)})$, et de plus $\exists x' \in V'$, $x \in \overrightarrow{E'}(x')$ et $\sigma_0(x') = p$. Ajoutons à x un nouveau fils étiqueté $(\sigma_0(x), \nu(x) - 1)$ et marqué, pour chaque x correspondant aux critères précédents. Dans la nouvelle Q-forêt ρ_0 ainsi obtenue, tous les nœuds non marqués satisfont tous les critères de la définition d'une exécution de $\mathcal{A}$ sur u. Nous allons construire par induction une suite $(\rho_n)_{n\geq 0}$ de Q-forêts conservant cette propriété, et n'ayant aucun nœud marqué de profondeur inférieure à n.

Supposons que nous ayons construit la Q-forêt ρ_n pour $n \geq 0$. Remplaçons dans ρ_n tout nœud marqué par une copie du sous-arbre de son grand-père (ce qui crée un nouveau nœud marqué à la profondeur $n + 2$). La nouvelle Q-forêt ρ_{n+1} ainsi obtenue satisfait bien nos attentes. Les exécutions ρ_n et ρ_{n+1} étant identiques jusqu'à la profondeur $n - 1$, la suite $(\rho_n)_{n\geq 0}$ converge vers une Q-forêt $\rho = (V, E, \sigma, \nu)$. Les racines de ρ sont les racines de ρ' et satisfont la condition initiale $I = I'$. De plus ρ ne contient aucun nœud marqué et satisfait donc la fonction de transition δ : ρ est une exécution de $\mathcal{A}$ sur u.

Pour tout $x \in V$, $\nu(x) \in \{0, |u| + 1\} \Rightarrow \sigma(x) \in F$. Soit $x_0, x_1, \ldots$ une branche infinie de ρ. Si $\exists N \geq 0$ tel que $\forall n \geq N$, x_n n'est pas marqué dans ρ_n, alors $x_0, x_1, \ldots$ est une branche de ρ_N. Elle finit donc comme une branche de ρ_0, et est ultimement étiquetée par un état de R. Sinon, $\forall N \geq 0$, $\exists n \geq N$ tel que x_n est marqué dans ρ_n. Tout nœud marqué possède une étiquette d'état dans R, et $\mathcal{A}$ est très faible donc cette branche est ultimement étiquetée par un état de R. Ainsi ρ est acceptante, et $u \in \mathcal{L}(\mathcal{A})$. □

8.3.3 Complexité

Pour tout automate alternant très faible à double sens $\mathcal{A}$ avec n états, l'automate $\mathcal{A}_p$ possède au maximum $2n$ états.

8.4 De PLTL aux automates progressants très faibles

Dans cette section nous exposons comment, à partir d'une formule PLTL φ de taille temporelle $|\varphi|$, construire un automate progressant très faible $\mathcal{A}_\varphi$ de même langage, avec au plus $|\varphi| + 1$ états.

8.4.1 Construction

Nous commençons par introduire deux notations utiles pour la construction.

Définition 8.8 (*notations*)

1. Pour $J_1, J_2 \in 2^{2^Q}$, on note $J_1 \otimes J_2 = \{X_1 \cup X_2 \mid X_1 \in J_1 \text{ et } X_2 \in J_2\}$.
2. De même, pour $J_1, J_2 \in 2^{2^Q \times 2^Q}$, on note
$J_1 \otimes J_2 = \{(\overleftarrow{X}_1 \cup \overleftarrow{X}_2, \overrightarrow{X}_1 \cup \overrightarrow{X}_2) \mid (\overleftarrow{X}_1, \overrightarrow{X}_1) \in J_1 \text{ et } (\overleftarrow{X}_2, \overrightarrow{X}_2) \in J_2\}$.
3. Pour toute formule ψ en forme normale négative, on définit $\overline{\psi}$ par induction :
   - $\overline{\psi} = \{\{\psi\}\}$ si ψ est une formule temporelle,
   - $\overline{\psi_1 \vee \psi_2} = \overline{\psi_1} \cup \overline{\psi_2}$,
   - $\overline{\psi_1 \wedge \psi_2} = \overline{\psi_1} \otimes \overline{\psi_2}$.

Contrairement au cas de LTL où les sous-formules de φ suffisent comme états de l'automate alternant, ici nous aurons besoin d'un état supplémentaire. Dans toute exécution acceptante sur un mot u, cet état, noté END, sera atteint quand la position courante sera en-dehors du mot u : $(\forall x \in V,\ \sigma(x) = \mathrm{END} \Rightarrow \nu(x) \in \{0, |u| + 1\})$. On définit $\overline{\mathrm{END}} = \{\{\mathrm{END}\}\}$.

Définition 8.9 ($\varphi \to \mathcal{A}_\varphi$)

Soient φ une formule PLTL sur l'ensemble AP, et $sub(\varphi)$ l'ensemble de ses sous-formules temporelles. L'automate $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, F, R)$ est défini ainsi :
- $Q = sub(\varphi) \cup \{\mathrm{END}\}$,
- $\Sigma = 2^{\mathrm{AP}}$,
- $I = \overline{\varphi}$,
- $F = \{\mathrm{END}\}$,
- R est l'ensemble des éléments de $sub(\varphi)$ qui ne sont pas de la forme $\psi_1 \mathsf{U} \psi_2$,
- δ est la restriction à $Q \times \Sigma$ de la fonction suivante, définie sur $\mathcal{B}^+(Q) \times \Sigma$:

$$\forall a \in \Sigma, \begin{cases} \delta(\mathrm{END}, a) = \emptyset \\ \delta(\bot, a) = \emptyset \\ \delta(\top, a) = \{(\emptyset, \emptyset)\} \\ \delta(p, a) = \{(\emptyset, \emptyset)\} \text{ si } p \in a, \emptyset \text{ sinon} \\ \delta(\neg p, a) = \{(\emptyset, \emptyset)\} \text{ si } p \notin a, \emptyset \text{ sinon} \\ \delta(\psi_1 \vee \psi_2, a) = \delta(\psi_1, a) \cup \delta(\psi_2, a) \\ \delta(\psi_1 \wedge \psi_2, a) = \delta(\psi_1, a) \otimes \delta(\psi_2, a) \\ \delta(\mathsf{X}\,\psi, a) = \{\emptyset\} \times \overline{\psi} \\ \delta(\widetilde{\mathsf{X}}\,\psi, a) = \{\emptyset\} \times (\overline{\psi} \cup \overline{\mathrm{END}}) \\ \delta(\mathsf{Y}\,\psi, a) = \overline{\psi} \times \{\emptyset\} \\ \delta(\widetilde{\mathsf{Y}}\,\psi, a) = (\overline{\psi} \cup \overline{\mathrm{END}}) \times \{\emptyset\} \\ \delta(\psi_1 \,\mathsf{U}\, \psi_2, a) = \delta(\psi_2, a) \cup (\delta(\psi_1, a) \otimes \{(\emptyset, \{\psi_1 \,\mathsf{U}\, \psi_2\})\}) \\ \delta(\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, a) = \delta(\psi_2, a) \otimes (\delta(\psi_1, a) \cup \{(\emptyset, \{\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2\}), (\emptyset, \{\mathrm{END}\})\}) \\ \delta(\psi_1 \,\mathsf{S}\, \psi_2, a) = \delta(\psi_2, a) \cup (\delta(\psi_1, a) \otimes \{(\{\psi_1 \,\mathsf{S}\, \psi_2\}, \emptyset)\}) \\ \delta(\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2, a) = \delta(\psi_2, a) \otimes (\delta(\psi_1, a) \cup \{(\{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2\}, \emptyset), (\{\mathrm{END}\}, \emptyset)\}) \end{cases}$$

Proposition 8.11

Pour toute formule PLTL φ, l'automate $\mathcal{A}_\varphi$ est un automate progressant très faible.

Démonstration. Définissons la relation $\psi_1 \preccurlyeq \psi_2$ si $\psi_1 = \text{END}$ ou si ψ_1 est une sous-formule de ψ_2. On vérifie aisément que $\preccurlyeq$ est un ordre partiel sur Q, et qu'il permet de prouver que $\mathcal{A}_\varphi$ est très faible.

Vérifions maintenant que $\mathcal{A}_\varphi$ est progressant. Supposons qu'un nœud x et son fils y possèdent la même étiquette d'état ψ sur une exécution de $\mathcal{A}_\varphi$. Dans ce cas, deux possibilités sont envisageables :

- $y \in \overrightarrow{E}(x)$ et ($\psi = \psi_1 \mathsf{U} \psi_2$ ou $\psi = \psi_1 \widetilde{\mathsf{U}} \psi_2$)
- $y \in \overleftarrow{E}(x)$ et ($\psi = \psi_1 \mathsf{S} \psi_2$ ou $\psi = \psi_1 \widetilde{\mathsf{S}} \psi_2$)

Soit $x_0, x_1, \ldots$ une branche infinie d'une exécution ρ de $\mathcal{A}_\varphi$. L'automate $\mathcal{A}_\varphi$ étant très faible, la suite $\sigma(x_0), \sigma(x_1), \ldots$ est ultimement constante. D'après le paragraphe précédent, cette limite est nécessairement de la forme $\psi_1 \mathsf{U} \psi_2$ ou $\psi_1 \widetilde{\mathsf{U}} \psi_2$, car ν est toujours minorée par 0. Ainsi, ν est ultimement strictement croissante, et $\mathcal{A}_\varphi$ est progressant. □

8.4.2 Preuve de l'équivalence

Comme pour le cas de LTL, nous allons prouver que $\mathcal{A}_\varphi$ reconnaît le langage de la formule φ, en démontrant au préalable quelques lemmes intermédiaires. Nous allons commencer par généraliser la notion d'exécution.

Définition 8.10 (*exécutions d'un automate alternant à double sens*)

Soient $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate alternant à double-sens, $J \subseteq 2^Q$, $i \in \mathbb{N}^*$. Soit $u = u_1 u_2 \ldots \in \Sigma^\infty$, avec $|u| \geq i$. Une (J, i)-exécution de l'automate $\mathcal{A}$ sur le mot u est une Q-forêt $\rho = (V, E, \sigma, \nu)$ telle que :

- Les racines satisfont la condition initiale : $\sigma(\Gamma) \in J$ et $\nu(\Gamma) = \{i\}$,
- Les fils d'un nœud satisfont la fonction de transition : $\forall x \in V$,
  - si $1 \leq \nu(x) \leq |u|$, $(\sigma(\overleftarrow{E}(x)), \sigma(\overrightarrow{E}(x))) \in \delta(\sigma(x), u_{\nu(x)})$.
  - sinon, $E(x) = \emptyset$.

Une (J, i)-exécution est *acceptante* si, de plus,

- $\forall x \in V$, si $\nu(x) = 0$ ou $\nu(x) = |u| + 1$, alors $\sigma(x) \in F$,
- chaque branche infinie de la Q-forêt ρ possède un nombre infini de nœuds étiquetés dans R.

Le langage $\mathcal{L}(\mathcal{A}, J, i)$ est l'ensemble des mots de Σ^∞ de longueur supérieure à i sur lesquels il existe une (J, i)-exécution acceptante de $\mathcal{A}$.

Remarque.

Une exécution de $\mathcal{A}$ sur un mot u, au sens défini dans la définition 8.4, est en fait une $(I, 1)$-exécution.

Lemme 8.12

Soient $\mathcal{A}$ un automate alternant à double sens, $J_1, J_2 \in 2^{2^Q}$, $i \in \mathbb{N}^*$:

1. $\mathcal{L}(\mathcal{A}, J_1 \cup J_2, i) = \mathcal{L}(\mathcal{A}, J_1, i) \cup \mathcal{L}(\mathcal{A}, J_2, i)$,
2. $\mathcal{L}(\mathcal{A}, J_1 \otimes J_2, i) = \mathcal{L}(\mathcal{A}, J_1, i) \cap \mathcal{L}(\mathcal{A}, J_2, i)$,

Démonstration. Soit $u \in \Sigma^\infty$:

1. $u \in \mathcal{L}(\mathcal{A}, J_1 \cup J_2, i)$
   ssi il existe une $(J_1 \cup J_2, i)$-exécution acceptante ρ de $\mathcal{A}$ sur u
   ssi il existe une (J_1, i)-exécution acceptante ρ_1 de $\mathcal{A}$ sur u ou
   il existe une (J_2, i)-exécution acceptante ρ_2 de $\mathcal{A}$ sur u
   ssi $u \in \mathcal{L}(\mathcal{A}, J_1, i)$ ou $u \in \mathcal{L}(\mathcal{A}, J_2, i)$
   ssi $u \in \mathcal{L}(\mathcal{A}, J_1, i) \cup \mathcal{L}(\mathcal{A}, J_2, i)$
   En effet il suffit, dans le sens direct de prendre $\rho_1 = \rho_2 = \rho$, et dans le sens indirect de prendre $\rho = \rho_1$ ou $\rho = \rho_2$.

2. La preuve de cette égalité est similaire. Dans le sens indirect, on prendra cette fois $\rho = \rho_1 \cup \rho_2$. $\sigma_1(\Gamma_1) \in J_1$ et $\sigma_2(\Gamma_2) \in J_2$ donc $\sigma(\Gamma) = \sigma(\Gamma_1) \cup \sigma(\Gamma_2) \in J_1 \otimes J_2$. □

Lemme 8.13

Soient φ une formule PLTL, et $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, F, R)$. Pour toute sous-formule ψ de φ, $\forall a \in \Sigma$, $\delta(\psi, a) = \bigcup_{X \in \overline{\psi}} \left(\bigotimes_{q \in X} \delta(q, a) \right)$. De même, $\delta(\mathrm{END}, a) = \bigcup_{X \in \overline{\mathrm{END}}} \left(\bigotimes_{q \in X} \delta(q, a) \right)$.

Démonstration. Nous allons prouver ce lemme par récurrence sur le nombre d'opérateurs $\vee$ et $\wedge$ de ψ.

- Si $\psi = \mathrm{END}$ ou si ψ est une sous-formule temporelle de φ, alors $\overline{\psi} = \{\{\psi\}\}$ donc $\delta(\psi, a) = \bigcup_{X \in \overline{\psi}} \left(\bigotimes_{q \in X} \delta(q, a) \right)$
- Si $\psi = \psi_1 \vee \psi_2$, alors on peut utiliser l'hypothèse de récurrence sur ψ_1 et ψ_2, et
$$\begin{aligned}\delta(\psi, a) &= \delta(\psi_1, a) \cup \delta(\psi_2, a) \\ &= \bigcup_{X \in \overline{\psi_1}} \left(\bigotimes_{q \in X} \delta(q, a) \right) \cup \bigcup_{X \in \overline{\psi_2}} \left(\bigotimes_{q \in X} \delta(q, a) \right) \\ &= \bigcup_{X \in \overline{\psi_1} \cup \overline{\psi_2}} \left(\bigotimes_{q \in X} \delta(q, a) \right) \\ &= \bigcup_{X \in \overline{\psi}} \left(\bigotimes_{q \in X} \delta(q, a) \right)\end{aligned}$$
- Si $\psi = \psi_1 \wedge \psi_2$, alors on peut utiliser l'hypothèse de récurrence sur ψ_1 et ψ_2, et

$$\begin{aligned}
\delta(\psi, a) &= \delta(\psi_1, a) \otimes \delta(\psi_2, a) \\
&= \bigcup_{X_1 \in \overline{\psi_1}} \Big(\bigotimes_{q \in X_1} \delta(q, a) \Big) \otimes \bigcup_{X_2 \in \overline{\psi_2}} \Big(\bigotimes_{q \in X_2} \delta(q, a) \Big) \\
&= \bigcup_{X_1 \in \overline{\psi_1}} \bigcup_{X_2 \in \overline{\psi_2}} \Big(\bigotimes_{q \in X_1} \delta(q, a) \otimes \bigotimes_{q \in X_2} \delta(q, a) \Big) \\
&= \bigcup_{X_1 \cup X_2 \in \overline{\psi_1} \otimes \overline{\psi_2}} \Big(\bigotimes_{q \in X_1 \cup X_2} \delta(q, a) \Big) \\
&= \bigcup_{X \in \overline{\psi}} \Big(\bigotimes_{q \in X} \delta(q, a) \Big)
\end{aligned}$$

□

Lemme 8.14

Soit φ une formule PLTL.
Pour toute sous-formule ψ de φ, pour tout $i \in \mathbb{N}^*$, $\mathcal{L}(\mathcal{A}_\varphi, \overline{\psi}, i) = \mathcal{L}(\psi, i)$.

Démonstration. Nous allons prouver cette affirmation par induction sur ψ. Soit $i \in \mathbb{N}^*$:

– $\overline{\bot} = \{\{\bot\}\}$ et $\forall a \in \Sigma$, $\delta(\bot, a) = \emptyset$. Une $(\overline{\bot}, i)$-exécution de $\mathcal{A}_\varphi$ sur un mot u doit contenir une racine d'étiquette $(\bot, i)$, or ce nœud ne peut pas satisfaire la fonction de transition. Aucun mot ne peut être accepté.
$\mathcal{L}(\mathcal{A}_\varphi, \overline{\bot}, i) = \emptyset = \mathcal{L}(\bot, i)$.

– $\overline{\top} = \{\{\top\}\}$ et $\forall a \in \Sigma$, $\delta(\top, a) = \{(\emptyset, \emptyset)\}$. La Q-forêt constituée d'une unique racine d'étiquette $(\top, i)$ est une $(\overline{\top}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur tout mot $u \in \Sigma^\infty$.
$\mathcal{L}(\mathcal{A}_\varphi, \overline{\top}, i) = \Sigma^\infty = \mathcal{L}(\top, i)$.

– $\overline{p} = \{\{p\}\}$ et $\forall a \in \Sigma$, $\delta(p, a) = \{(\emptyset, \emptyset)\}$ si $p \in a$, $\emptyset$ sinon. Une $(\overline{p}, i)$-exécution de $\mathcal{A}_\varphi$ sur un mot $u = u_1 u_2 \ldots$ doit contenir une racine d'étiquette (p, i). Si $a \in u_i$ alors cette $(\overline{p}, i)$-exécution est acceptante, sinon la fonction de transition ne peut pas être satisfaite.
$\mathcal{L}(\mathcal{A}_\varphi, \overline{p}, i) = \{u = u_1 u_2 \ldots \in \Sigma^\infty \mid 1 \leq i \leq |u| \text{ et } p \in u_i\} = \mathcal{L}(p, i)$.

– de même,
$\mathcal{L}(\mathcal{A}_\varphi, \overline{\neg p}, i) = \{u = u_1 u_2 \ldots \in \Sigma^\infty \mid 1 \leq i \leq |u| \text{ et } p \notin u_i\} = \mathcal{L}(\neg p, i)$.

– $$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \vee \psi_2}, i) &= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1} \cup \overline{\psi_2}, i) && (\textit{définition 8.8.3}) \\
&= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, i) \cup \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, i) && (\textit{lemme 8.12.1}) \\
&= \mathcal{L}(\psi_1, i) \cup \mathcal{L}(\psi_2, i) && (\textit{hypothèse d'induction}) \\
&= \mathcal{L}(\psi_1 \vee \psi_2, i) && (\textit{sémantique de LTL})
\end{aligned}$$

– $$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \wedge \psi_2}, i) &= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1} \otimes \overline{\psi_2}, i) && (\textit{définition 8.8.3}) \\
&= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, i) \cap \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, i) && (\textit{lemme 8.12.2}) \\
&= \mathcal{L}(\psi_1, i) \cap \mathcal{L}(\psi_2, i) && (\textit{hypothèse d'induction}) \\
&= \mathcal{L}(\psi_1 \wedge \psi_2, i) && (\textit{sémantique de LTL})
\end{aligned}$$

– $\overline{\mathsf{X}\,\psi} = \{\{\mathsf{X}\,\psi\}\}$ et $\forall a \in \Sigma$, $\delta(\mathsf{X}\,\psi, a) = \{\emptyset\} \times \overline{\psi}$.

Soit ρ une $(\overline{\mathsf{X}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. ρ contient une racine x d'étiquette $(\mathsf{X}\,\psi, i)$, telle que $\overleftarrow{E}(x) = \emptyset$ et $\sigma(\overrightarrow{E}(x)) \in \overline{\psi}$. Ainsi $i < |u|$ et la Q-forêt de racines $\overrightarrow{E}(x)$ est une $(\overline{\psi}, i+1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u.

Réciproquement, étant donné une $(\overline{\psi}, i+1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u, en ajoutant un père commun d'étiquette $(\mathsf{X}\,\psi, i)$ à toutes les racines, on obtient clairement une $(\overline{\mathsf{X}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u.

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \overline{\mathsf{X}\,\psi}, i) &= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi}, i+1) \\
&= \mathcal{L}(\psi, i+1) && (\textit{hypothèse d'induction}) \\
&= \mathcal{L}(\mathsf{X}\,\psi, i) && (\textit{sémantique de LTL})
\end{aligned}$$

– $\overline{\widetilde{\mathsf{X}}\,\psi} = \{\{\widetilde{\mathsf{X}}\,\psi\}\}$ et $\forall a \in \Sigma$, $\delta(\widetilde{\mathsf{X}}\,\psi, a) = \{\emptyset\} \times (\overline{\psi} \cup \overline{\mathrm{END}})$.

Soit ρ une $(\overline{\widetilde{\mathsf{X}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. ρ contient une racine x d'étiquette $(\widetilde{\mathsf{X}}\,\psi, i)$, telle que $\overleftarrow{E}(x) = \emptyset$ et $\sigma(\overrightarrow{E}(x)) \in \overline{\psi}$ ou $\sigma(\overrightarrow{E}(x)) \in \overline{\mathrm{END}}$. Dans le premier cas, $i < |u|$ et la Q-forêt de racines $\overrightarrow{E}(x)$ est une $(\overline{\psi}, i+1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u. Dans le deuxième cas, ρ étant acceptante, u est fini et $i = |u|$.

Réciproquement, étant donné une $(\overline{\psi}, i+1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u, en ajoutant un père commun d'étiquette $(\widetilde{\mathsf{X}}\,\psi, i)$ à toutes les racines, on obtient clairement une $(\overline{\widetilde{\mathsf{X}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u. D'autre part, pour tout mot fini u de longueur i, la Q-forêt composée d'une racine d'étiquette $(\widetilde{\mathsf{X}}\,\psi, |u|)$ avec un fils d'étiquette $(\mathrm{END}, |u|+1)$ est aussi une $(\overline{\widetilde{\mathsf{X}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u.

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \overline{\widetilde{\mathsf{X}}\,\psi}, i) &= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi}, i+1) \cup \Sigma^i \\
&= \mathcal{L}(\psi, i+1) \cup \Sigma^i && (\textit{hypothèse d'induction}) \\
&= \mathcal{L}(\widetilde{\mathsf{X}}\,\psi, i) && (\textit{sémantique de LTL})
\end{aligned}$$

– $\overline{\mathsf{Y}\,\psi} = \{\{\mathsf{Y}\,\psi\}\}$ et $\forall a \in \Sigma$, $\delta(\mathsf{Y}\,\psi, a) = \overline{\psi} \times \{\emptyset\}$.

Soit ρ une $(\overline{\mathsf{Y}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. ρ contient une racine x d'étiquette $(\mathsf{Y}\,\psi, i)$, telle que $\overrightarrow{E}(x) = \emptyset$ et $\sigma(\overleftarrow{E}(x)) \in \overline{\psi}$. Ainsi $i > 1$ et la Q-forêt de racines $\overleftarrow{E}(x)$ est une $(\overline{\psi}, i-1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u.

Réciproquement, si $i > 1$, étant donné une $(\overline{\psi}, i-1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u, en ajoutant un père commun d'étiquette $(\mathsf{Y}\,\psi, i)$ à toutes les racines, on obtient clairement une $(\overline{\mathsf{Y}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u.

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_\varphi, \overline{\mathsf{Y}\,\psi}, i) &= \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi}, i-1) \\
&= \mathcal{L}(\psi, i-1) && (\textit{hypothèse d'induction}) \\
&= \mathcal{L}(\mathsf{Y}\,\psi, i) && (\textit{sémantique de LTL})
\end{aligned}$$

– $\overline{\widetilde{\mathsf{Y}}\,\psi} = \{\{\widetilde{\mathsf{Y}}\,\psi\}\}$ et $\forall a \in \Sigma$, $\delta(\widetilde{\mathsf{Y}}\,\psi, a) = (\overline{\psi} \cup \overline{\mathrm{END}}) \times \{\emptyset\}$.

Soit ρ une $(\overline{\widetilde{\mathsf{Y}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. ρ contient une racine x d'étiquette $(\widetilde{\mathsf{Y}}\,\psi, i)$, telle que $\overrightarrow{E}(x) = \emptyset$ et $\sigma(\overleftarrow{E}(x)) \in \overline{\psi}$ ou $\sigma(\overleftarrow{E}(x)) \in \overline{\mathrm{END}}$. Dans le premier cas, $i > 1$ et la Q-forêt de racines $\overleftarrow{E}(x)$ est une $(\overline{\psi}, i-1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u. Dans le deuxième cas, ρ étant acceptante, $i = 1$.

Réciproquement, étant donné une $(\overline{\psi}, i-1)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u, en ajoutant un père commun d'étiquette $(\widetilde{\mathsf{Y}}\,\psi, i)$ à toutes les racines, on obtient clairement une $(\overline{\widetilde{\mathsf{Y}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur u. D'autre part, si $i = 1$, la Q-forêt composée d'une racine d'étiquette $(\widetilde{\mathsf{X}}\,\psi, 1)$ avec un fils d'étiquette $(\mathrm{END}, 0)$ est aussi une $(\overline{\widetilde{\mathsf{X}}\,\psi}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur tout mot u.

Si $i > 1$, $\mathcal{L}(\mathcal{A}_\varphi, \overline{\widetilde{\mathsf{Y}}\,\psi}, i) = \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi}, i-1)$
$= \mathcal{L}(\psi, i-1)$ (*hypothèse d'induction*)
$= \mathcal{L}(\widetilde{\mathsf{Y}}\,\psi, i)$ (*sémantique de LTL*)

Si $i = 1$, $\mathcal{L}(\mathcal{A}_\varphi, \overline{\widetilde{\mathsf{Y}}\,\psi}, 1) = \Sigma^\infty = \mathcal{L}(\widetilde{\mathsf{Y}}\,\psi, 1)$

– $\overline{\psi_1 \,\mathsf{U}\, \psi_2} = \{\{\psi_1 \mathsf{U} \psi_2\}\}$ et $\forall a \in \Sigma$, $\delta(\psi_1 \mathsf{U} \psi_2, a) = \delta(\psi_2, a) \cup (\delta(\psi_1, a) \otimes \{(\emptyset, \{\psi_1 \mathsf{U} \psi_2\})\})$.
Soit $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \,\mathsf{U}\, \psi_2}, i)$, et soit $\rho = (V, E, \sigma, \nu)$ une $(\overline{\psi_1 \,\mathsf{U}\, \psi_2}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. La Q-forêt ρ contient une racine x_i d'étiquette $(\psi_1 \mathsf{U} \psi_2, i)$, telle que $(\sigma(\overleftarrow{E}(x_i)), \sigma(\overrightarrow{E}(x_i))) \in \delta(\psi_2, u_i)$ ou bien $i < |u|$ et x_i possède un fils x_{i+1} d'étiquette $(\psi_1 \mathsf{U} \psi_2, i+1)$ et $(\sigma(\overleftarrow{E}(x_i)), \sigma(\overrightarrow{E}(x_i) \backslash \{x_{i+1}\})) \in \delta(\psi_1, u_i)$. En itérant cette propriété, on constate que x_i est la tête d'une chaîne $x_i, x_{i+1}, \ldots, x_k$, avec $k \leq |u|$ telle que $\forall i \leq j \leq k$, $\lambda(x_j) = (\psi_1 \,\mathsf{U}\, \psi_2, j)$, $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \delta(\psi_2, u_k)$ et $\forall i \leq j < k$, $(x_j, x_{j+1}) \in E$ et $(\sigma(\overleftarrow{E}(x_j)), \sigma(\overrightarrow{E}(x_j) \backslash \{x_{j+1}\})) \in \delta(\psi_1, u_j)$. On notera que la chaîne est nécessairement finie, sinon ρ contiendrait une branche infinie ultimement étiquetée par $\psi_1 \,\mathsf{U}\, \psi_2 \notin R$.
D'après le lemme 8.13, $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \delta(\psi_2, u_k)$ donc $\exists \Gamma_k \in \overline{\psi_2}$ tel que $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \bigotimes_{q \in \Gamma_k} \delta(q, u_i)$. Ainsi $\sigma(\overleftarrow{E}(x_k)) = \bigcup_{q \in \Gamma_k} \overleftarrow{Y}_q$ et $\sigma(\overrightarrow{E}(x_k)) = \bigcup_{q \in \Gamma_k} \overrightarrow{Y}_q$ avec $\forall q \in \Gamma_k$, $(\overleftarrow{Y}_q, \overrightarrow{Y}_q) \in \delta(q, u_i)$. Nous allons définir une Q-forêt $\rho_k = (V_k, E_k, \sigma_k, \nu_k)$ de la façon suivante :
– $V_k = E^+(x_k) \cup \Gamma_k$,
– $E_k = E_{|E^+(x_k) \times E^+(x_k)} \cup \{(q, y) \mid q \in \Gamma_k\ ,\ y \in \overleftarrow{E}(x_k)$ et $\sigma(y) \in \overleftarrow{Y}_q$
ou $y \in \overrightarrow{E}(x_k)$ et $\sigma(y) \in \overrightarrow{Y}_q\}$,
– $\sigma_k(x) = x$ si $x \in \Gamma_k$, $\sigma(x)$ sinon,
– $\nu_k(x) = k$ si $x \in \Gamma_k$, $\nu(x)$ sinon.
Par construction, Γ_k est l'ensemble des racines de ρ_k, $\sigma(\Gamma_k) \in \overline{\psi_2}$, $\nu(\Gamma_k) = \{k\}$ et $(\sigma(\overleftarrow{E}(\Gamma_k)), \sigma(\overrightarrow{E}(\Gamma_k))) \in \bigotimes_{q \in \Gamma_k} \delta(q, u_i)$. Ainsi la Q-forêt ρ_k est une $(\overline{\psi_2}, k)$-exécution acceptante de $\mathcal{A}$ sur u.

De même, grâce au lemme 8.13, $\forall i \leq j < k$, on peut obtenir une Q-forêt $\rho_j = (V_j, E_j, \sigma_j, \nu_j)$, $(\overline{\psi_1}, j)$-exécution acceptante de $\mathcal{A}$ sur u.
Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall i \leq j < k$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$.

Réciproquement, considérons un mot u et un entier k avec $i \leq k \leq |u|$, tels que $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall i \leq j < k$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$. Il existe une $(\overline{\psi_2}, k)$-exécution acceptante ρ_k de $\mathcal{A}$ sur u, et $\forall i \leq j < k$, il existe une $(\overline{\psi_1}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_i, \ldots, x_k$ des nouveaux nœuds. On définit alors :
- $V = \bigcup_{i \leq j \leq k} V_j \backslash \Gamma_j \cup \{x_j \mid i \leq j \leq k\}$,
- $E = \bigcup_{i \leq j \leq k} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{i \leq j \leq k} (\{x_j\} \times E_j(\Gamma_j)) \cup \bigcup_{i \leq j < k} (x_j, x_{j+1})$,
- $\forall i \leq j \leq k$, $\lambda(x_j) = (\psi_1 \mathsf{U}\, \psi_2, j)$ et $\forall x \in V_j \backslash \Gamma_j$, $\lambda(x) = \lambda_j(x)$.

On constate aisément que ρ est une $(\overline{\psi_1 \mathsf{U}\, \psi_2}, i)$-exécution de $\mathcal{A}$ sur u. Toute branche infinie de ρ terminant comme une branche infinie de l'une des ρ_j, ρ est nécessairement acceptante, et donc $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \mathsf{U}\, \psi_2}, i)$.

Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \mathsf{U}\, \psi_2}, i)$
ssi $\exists i \leq k \leq |u|$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall i \leq j < k$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$
ssi $\exists i \leq k \leq |u|$, $u \in \mathcal{L}(\psi_2, k)$ et $\forall i \leq j < k$, $u \in \mathcal{L}(\psi_1, j)$ (*hyp. d'induction*)
ssi $u \in \mathcal{L}(\psi_1 \mathsf{U}\, \psi_2, i)$ (*sémantique de LTL*)

– $\overline{\psi_1 \widetilde{\mathsf{U}}\, \psi_2} = \{\{\psi_1 \widetilde{\mathsf{U}}\, \psi_2\}\}$
et $\forall a \in \Sigma$, $\delta(\psi_1 \widetilde{\mathsf{U}}\, \psi_2, a) = \delta(\psi_2, a) \otimes (\delta(\psi_1, a) \cup \{(\emptyset, \{\psi_1 \widetilde{\mathsf{U}}\, \psi_2\}), (\emptyset, \{\mathrm{END}\})\})$.
Soit $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \widetilde{\mathsf{U}}\, \psi_2}, i)$, et soit $\rho = (V, E, \sigma, \nu)$ une $(\overline{\psi_1 \widetilde{\mathsf{U}}\, \psi_2}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. ρ contient une racine x_i d'étiquette $(\psi_1 \widetilde{\mathsf{U}}\, \psi_2, i)$, telle que $\overleftarrow{E}(x_i) = \overleftarrow{X}_1 \cup \overleftarrow{X}_2$ et $\overrightarrow{E}(x_i) = \overrightarrow{X}_1 \cup \overrightarrow{X}_2$, avec $(\sigma(\overleftarrow{X}_2), \sigma(\overrightarrow{X}_2)) \in \delta(\psi_2, u_i)$ et ou bien $(\sigma(\overleftarrow{X}_1), \sigma(\overrightarrow{X}_1)) \in \delta(\psi_1, u_i)$, ou bien $i < |u|$ et $\sigma(\overleftarrow{X}_1) = \emptyset$ et $\sigma(\overrightarrow{X}_1) = \{\psi_1 \widetilde{\mathsf{U}} \psi_2\}$, ou enfin u est fini, $i = |u|$ et $\sigma(\overleftarrow{X}_1) = \emptyset$ et $\sigma(\overrightarrow{X}_1) = \{\mathrm{END}\}$. Ainsi x_i est la tête d'une chaîne finie ou infinie $x_i, x_{i+1}, \ldots$ telle que $\forall j \geq i$, $\lambda(x_j) = (\psi_1 \widetilde{\mathsf{U}} \psi_2, j)$, et si x_j n'est pas le dernier nœud de la chaîne, $(x_j, x_{j+1}) \in E$ et $(\sigma(\overleftarrow{E}(x_j)), \sigma(\overrightarrow{E}(x_j) \backslash \{x_{j+1}\})) \in \delta(\psi_2, u_j)$. De plus si la chaîne est finie, soit x_k son dernier nœud : ou bien $|u| = k$ et x_k possède un fils droit x' d'étiquette d'état END et $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k) \backslash \{x'\})) \in \delta(\psi_2, u_k)$, ou bien $\overleftarrow{E}(x_k) = \overleftarrow{X}_1 \cup \overleftarrow{X}_2$ et $\overrightarrow{E}(x_k) = \overrightarrow{X}_1 \cup \overrightarrow{X}_2$, avec $(\sigma(\overleftarrow{X}_2), \sigma(\overrightarrow{X}_2)) \in \delta(\psi_2, u_k)$ et $(\sigma(\overleftarrow{X}_1), \sigma(\overrightarrow{X}_1)) \in \delta(\psi_1, u_k)$.
Grâce au lemme 8.13, $\forall j \geq i$, on peut obtenir une Q-forêt $\rho_j = (V_j, E_j, \sigma_j, \nu_j)$, $(\overline{\psi_2}, j)$-exécution acceptante de $\mathcal{A}$ sur u. De plus, si la chaîne est finie et $\mathrm{END} \notin \sigma(\overrightarrow{E}(x_k))$, on peut obtenir une Q-forêt $\rho_{k+1} = (V_{k+1}, E_{k+1}, \sigma_{k+1}, \nu_{k+1})$, $(\overline{\psi_1}, k)$-exécution acceptante de $\mathcal{A}$ sur u. Ainsi deux cas sont possibles :
– $\forall j \in \mathbb{N}$, si $i \leq j \leq |u|$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,
– $\exists k \in \mathbb{N}$ tel que $i \leq k \leq |u|$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall i \leq j \leq k$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,

Réciproquement, considérons un mot u dans l'un des deux cas ci-dessus.

– Si u est infini et $\forall j \geq i,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$, alors pour chaque j il existe une $(\overline{\psi_2}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_i, x_{i+1}, \dots$ des nouveaux nœuds. On définit alors :
  - $V = \bigcup_{j\geq i} V_j \backslash \Gamma_j \cup \{x_j \mid j \geq i\}$,
  - $E = \bigcup_{j\geq i} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{j\geq i} (\{x_j\} \times E_j(\Gamma_j)) \cup \bigcup_{j\geq i} (x_j, x_{j+1})$,
  - $\forall j \geq i,\ \lambda(x_j) = (\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, j)$ et $\forall x \in V_j \backslash \Gamma_j,\ \lambda(x) = \lambda_j(x)$.

– Si u est fini et $\forall i \leq j \leq |u|,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$, alors pour chaque j il existe une $(\overline{\psi_2}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_i, \dots, x_{|u|+1}$ des nouveaux nœuds. On définit alors :
  - $V = \bigcup_{i\leq j\leq |u|} V_j \backslash \Gamma_j \cup \{x_j \mid i \leq j \leq |u|+1\}$,
  - $E = \bigcup_{i\leq j\leq |u|} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{i\leq j\leq |u|} (\{x_j\} \times E_j(\Gamma_j)) \cup \bigcup_{i\leq j\leq |u|} (x_j, x_{j+1})$,
  - $\forall i \leq j \leq |u|,\ \lambda(x_j) = (\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, j)$, $\lambda(x_{|u|+1}) = (\mathrm{END}, |u|+1)$, et $\forall x \in V_j \backslash \Gamma_j$, $\lambda(x) = \lambda_j(x)$.

– Si $\exists k \in \mathbb{N}$ tel que $i \leq k \leq |u|,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall i \leq j \leq k,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$, alors il existe une $(\overline{\psi_1}, k)$-exécution acceptante ρ_{k+1} de $\mathcal{A}$ sur u, et $\forall i \leq j \leq k$, il existe une $(\overline{\psi_2}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_i, \dots, x_k$ des nouveaux nœuds. On définit alors :
  - $V = \bigcup_{i\leq j\leq k+1} V_j \backslash \Gamma_j \cup \{x_j \mid i \leq j \leq k\}$,
  - $E = \bigcup_{i\leq j\leq k+1} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{i\leq j\leq k} (\{x_j\} \times E_j(\Gamma_j)) \cup \{x_k\} \times E_{k+1}(\Gamma_{k+1}) \cup \bigcup_{i\leq j<k} (x_j, x_{j+1})$,
  - $\forall i \leq j \leq k,\ \lambda(x_j) = (\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, j)$ et $\forall i \leq j \leq k+1,\ \forall x \in V_j \backslash \Gamma_j,\ \lambda(x) = \lambda_j(x)$.

Dans chacun des cas, on constate aisément que ρ est une $(\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, i)$-exécution de $\mathcal{A}$ sur u. Toute branche infinie de ρ termine comme une branche infinie de l'une des ρ_j, exceptée, pour le premier cas, la branche des x_j qui est étiquetée dans R, et dans le deuxième cas, celle qui se termine par un nœud d'étiquette $(\mathrm{END}, |u|+1)$. Ainsi ρ est nécessairement acceptante, et donc $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2}, i)$.

Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, i})$
ssi $\forall i \leq j \leq |u|,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,
  ou $\exists i \leq k \leq |u|,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall i \leq j \leq k,\ u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$
ssi $\forall i \leq j \leq |u|,\ u \in \mathcal{L}(\psi_2, j)$,
  ou $\exists i \leq k \leq |u|,\ u \in \mathcal{L}(\psi_1, k)$ et $\forall i \leq j \leq k,\ u \in \mathcal{L}(\psi_2, j)$ (*h. d'induction*)
ssi $u \in \mathcal{L}(\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2, i)$ (*sémantique de LTL*)

– $\overline{\psi_1 \mathsf{S} \psi_2} = \{\{\psi_1 \mathsf{S} \psi_2\}\}$ et $\forall a \in \Sigma$, $\delta(\psi_1 \mathsf{S} \psi_2, a) = \delta(\psi_2, a) \cup (\delta(\psi_1, a) \otimes \{(\{\psi_1 \mathsf{S} \psi_2\}, \emptyset)\})$.

Soit $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \mathsf{S} \psi_2}, i)$, et soit $\rho = (V, E, \sigma, \nu)$ une $(\overline{\psi_1 \mathsf{S} \psi_2}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. La Q-forêt ρ contient une racine x_i d'étiquette $(\psi_1 \mathsf{S} \psi_2, i)$, telle que $(\sigma(\overleftarrow{E}(x_i)), \sigma(\overrightarrow{E}(x_i))) \in \delta(\psi_2, u_i)$ ou bien $i > 1$ et x_i possède un fils x_{i-1} d'étiquette $(\psi_1 \mathsf{S} \psi_2, i-1)$ et $(\sigma(\overleftarrow{E}(x_i)), \sigma(\overrightarrow{E}(x_i) \backslash \{x_{i-1}\})) \in \delta(\psi_1, u_i)$. En itérant cette propriété, on constate que x_i est la tête d'une chaîne $x_i, x_{i-1}, \ldots, x_k$, avec $k \geq 1$ telle que $\forall k \leq j \leq i$, $\lambda(x_j) = (\psi_1 \mathsf{S} \psi_2, j)$, $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \delta(\psi_2, u_k)$ et $\forall k < j \leq i$, $(x_j, x_{j-1}) \in E$ et $(\sigma(\overleftarrow{E}(x_j)), \sigma(\overrightarrow{E}(x_j) \backslash \{x_{j-1}\})) \in \delta(\psi_1, u_j)$. On notera que la chaîne est nécessairement finie, car ν est minorée par 0.

D'après le lemme 8.13, $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \delta(\psi_2, u_k)$ donc $\exists \Gamma_k \in \overline{\psi_2}$ tel que $(\sigma(\overleftarrow{E}(x_k)), \sigma(\overrightarrow{E}(x_k))) \in \bigotimes_{q \in \Gamma_k} \delta(q, u_i)$. Ainsi $\sigma(\overleftarrow{E}(x_k)) = \bigcup_{q \in \Gamma_k} \overleftarrow{Y}_q$ et $\sigma(\overrightarrow{E}(x_k)) = \bigcup_{q \in \Gamma_k} \overrightarrow{Y}_q$ avec $\forall q \in \Gamma_k$, $(\overleftarrow{Y}_q, \overrightarrow{Y}_q) \in \delta(q, u_i)$. Nous allons définir une Q-forêt $\rho_k = (V_k, E_k, \sigma_k, \nu_k)$ de la façon suivante :

– $V_k = E^+(x_k) \cup \Gamma_k$,
– $E_k = E_{|E^+(x_k) \times E^+(x_k)} \cup \{(q, y) \mid q \in \Gamma_k\ ,\ y \in \overleftarrow{E}(x_k) \text{ et } \sigma(y) \in \overleftarrow{Y}_q$ ou $y \in \overrightarrow{E}(x_k) \text{ et } \sigma(y) \in \overrightarrow{Y}_q\}$,
– $\sigma_k(x) = x$ si $x \in \Gamma_k$, $\sigma(x)$ sinon,
– $\nu_k(x) = k$ si $x \in \Gamma_k$, $\nu(x)$ sinon.

Par construction, Γ_k est l'ensemble des racines de ρ_k, $\sigma(\Gamma_k) \in \overline{\psi_2}$, $\nu(\Gamma_k) = \{k\}$, et $(\sigma(\overleftarrow{E}(\Gamma_k)), \sigma(\overrightarrow{E}(\Gamma_k))) \in \bigotimes_{q \in \Gamma_k} \delta(q, u_i)$. Ainsi la Q-forêt ρ_k est une $(\overline{\psi_2}, k)$-exécution acceptante de $\mathcal{A}$ sur u.

De même, grâce au lemme 8.13, $\forall k < j \leq i$, on peut obtenir une Q-forêt $\rho_j = (V_j, E_j, \sigma_j, \nu_j)$, $(\overline{\psi_1}, j)$-exécution acceptante de $\mathcal{A}$ sur u.

Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall k < j \leq i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$.

Réciproquement, considérons un mot u et un entier k avec $1 \leq k \leq i$, tels que $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall k < j \leq i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$. Il existe une $(\overline{\psi_2}, k)$-exécution acceptante ρ_k de $\mathcal{A}$ sur u, et $\forall k < j \leq i$, il existe une $(\overline{\psi_1}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_k, \ldots, x_i$ des nouveaux nœuds. On définit alors :

- $V = \bigcup_{k \leq j \leq i} V_j \backslash \Gamma_j \cup \{x_j \mid k \leq j \leq i\}$,
- $E = \bigcup_{k \leq j \leq i} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{k \leq j \leq i} (\{x_j\} \times E_j(\Gamma_j)) \cup \bigcup_{k < j \leq i} (x_j, x_{j-1})$,
- $\forall k \leq j \leq i$, $\lambda(x_j) = (\psi_1 \mathsf{S} \psi_2, j)$ et $\forall x \in V_j \backslash \Gamma_j$, $\lambda(x) = \lambda_j(x)$.

On constate aisément que ρ est une $(\overline{\psi_1 \mathsf{S} \psi_2}, i)$-exécution de $\mathcal{A}$ sur u. Toute branche infinie de ρ terminant comme une branche infinie de l'une des ρ_j, ρ est nécessairement acceptante, et donc $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \mathsf{S} \psi_2}, i)$.

Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \,\mathsf{S}\, \psi_2}, i)$
ssi $\exists 1 \le k \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, k)$ et $\forall k < j \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, j)$
ssi $\exists 1 \le k \le i$, $u \in \mathcal{L}(\psi_2, k)$ et $\forall k < j \le i$, $u \in \mathcal{L}(\psi_1, j)$ (*h. d'induction*)
ssi $u \in \mathcal{L}(\psi_1 \,\mathsf{S}\, \psi_2, i)$ (*sémantique de LTL*)

– $\overline{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2} = \{\{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2\}\}$
et $\forall a \in \Sigma$, $\delta(\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2, a) = \delta(\psi_2, a) \otimes (\delta(\psi_1, a) \cup \{(\{\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2\}, \emptyset), (\{\mathrm{END}\}, \emptyset)\})$.

Soit $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2}, i)$, et soit $\rho = (V, E, \sigma, \nu)$ une $(\overline{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2}, i)$-exécution acceptante de $\mathcal{A}_\varphi$ sur un mot u. La Q-forêt ρ contient une racine x_i d'étiquette $(\psi_1 \widetilde{\mathsf{S}} \psi_2, i)$, telle que $\overleftarrow{E}(x_i) = \overleftarrow{X}_1 \cup \overleftarrow{X}_2$ et $\overrightarrow{E}(x_i) = \overrightarrow{X}_1 \cup \overrightarrow{X}_2$, avec $(\sigma(\overleftarrow{X}_2), \sigma(\overrightarrow{X}_2)) \in \delta(\psi_2, u_i)$ et ou bien $(\sigma(\overleftarrow{X}_1), \sigma(\overrightarrow{X}_1)) \in \delta(\psi_1, u_i)$, ou bien $i > 1$, $\sigma(\overleftarrow{X}_1) = \{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2\}$ et $\sigma(\overrightarrow{X}_1) = \emptyset$, ou enfin $i = 0$, $\sigma(\overleftarrow{X}_1) = \{\mathrm{END}\}$ et $\sigma(\overrightarrow{X}_1) = \emptyset$. Ainsi x_i est la tête d'une chaîne de nœuds $x_i, \dots, x_k$ telle que $\forall k \le j \le i$, $\lambda(x_j) = (\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2, j)$, et si $j > k$, $(x_j, x_{j+1}) \in E$ et $(\sigma(\overleftarrow{E}(x_j) \backslash \{x_{j+1}\}), \sigma(\overrightarrow{E}(x_j))) \in \delta(\psi_2, u_j)$, et de plus, ou bien ou bien $k = 1$ et x_k possède un fils gauche x' d'étiquette d'état END et $(\sigma(\overleftarrow{E}(x_k) \backslash \{x'\}), \sigma(\overrightarrow{E}(x_k))) \in \delta(\psi_2, u_k)$, ou bien $\overleftarrow{E}(x_k) = \overleftarrow{X}_1 \cup \overleftarrow{X}_2$ et $\overrightarrow{E}(x_k) = \overrightarrow{X}_1 \cup \overrightarrow{X}_2$, avec $(\sigma(\overleftarrow{X}_2), \sigma(\overrightarrow{X}_2)) \in \delta(\psi_2, u_k)$ et $(\sigma(\overleftarrow{X}_1), \sigma(\overrightarrow{X}_1)) \in \delta(\psi_1, u_k)$.

Grâce au lemme 8.13, $\forall k \le j \le i$, on peut obtenir une Q-forêt $\rho_j = (V_j, E_j, \sigma_j, \nu_j)$, $(\overline{\psi_2}, j)$-exécution acceptante de $\mathcal{A}$ sur u. De plus, si END $\notin \sigma(\overleftarrow{E}(x_k))$, on peut obtenir une Q-forêt $\rho_{k-1} = (V_{k-1}, E_{k-1}, \sigma_{k-1}, \nu_{k-1})$, $(\overline{\psi_1}, k)$-exécution acceptante de $\mathcal{A}$ sur u.

Ainsi deux cas sont possibles :

– $\forall 1 \le j \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,
– $\exists 1 \le k \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall k \le j \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,

Réciproquement, considérons un mot u dans l'un des deux cas ci-dessus.

– Si $\forall 1 \le j \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$, alors pour chaque j il existe une $(\overline{\psi_2}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_0, \dots, x_i$ des nouveaux nœuds. On définit alors :
  • $V = \bigcup_{1 \le j \le i} V_j \backslash \Gamma_j \cup \{x_j \mid 0 \le j \le i\}$,
  • $E = \bigcup_{1 \le j \le i} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{1 \le j \le i} (\{x_j\} \times E_j(\Gamma_j)) \cup \bigcup_{1 \le j \le i} (x_j, x_{j-1})$,
  • $\forall 1 \le j \le i$, $\lambda(x_j) = (\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2, j)$, $\lambda(x_0) = (\mathrm{END}, 0)$, et $\forall x \in V_j \backslash \Gamma_j$, $\lambda(x) = \lambda_j(x)$.
– Si $\exists 1 \le k \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall k \le j \le i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$, alors il existe une $(\overline{\psi_1}, k)$-exécution acceptante ρ_{k-1} de $\mathcal{A}$ sur u, et $\forall i \le j \le k$, il existe une $(\overline{\psi_2}, j)$-exécution acceptante ρ_j de $\mathcal{A}$ sur u. Nous allons maintenant construire une nouvelle Q-forêt ρ de la façon suivante : soient $x_k, \dots, x_i$ des nouveaux nœuds. On définit alors :

- $V = \bigcup_{k-1 \leq j \leq i} V_j \backslash \Gamma_j \cup \{x_j \mid k \leq j \leq i\}$,
- $E = \bigcup_{k-1 \leq j \leq i} E_j \backslash (\Gamma_j \times E_j(\Gamma_j)) \cup \bigcup_{k \leq j \leq i} (\{x_j\} \times E_j(\Gamma_j)) \cup \{x_k\} \times E_{k-1}(\Gamma_{k-1}) \cup \bigcup_{k < j \leq i} (x_j, x_{j-1})$,
- $\forall k \leq j \leq i$, $\lambda(x_j) = (\psi_1 \widetilde{\mathsf{S}}\, \psi_2, j)$ et $\forall k-1 \leq j \leq i$, $\forall x \in \underline{V_j \backslash \Gamma_j}$, $\lambda(x) = \lambda_j(x)$.

Dans chacun des cas, on constate aisément que ρ est une $(\psi_1 \widetilde{\mathsf{S}}\, \psi_2, i)$-exécution de $\mathcal{A}$ sur u. Toute branche infinie de ρ termine comme une branche infinie de l'une des ρ_j, exceptée, dans le premier cas, la branche qui se termine par un nœud d'étiquette $(\text{END}, 0)$. Ainsi ρ est nécessairement acceptante, et donc $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \widetilde{\mathsf{S}}\, \psi_2}, i)$.

Ainsi $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1 \widetilde{\mathsf{S}}\, \psi_2}, i)$
ssi $\forall 1 \leq j \leq i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$,
ou $\exists 1 \leq k \leq i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_1}, k)$ et $\forall k \leq j \leq i$, $u \in \mathcal{L}(\mathcal{A}_\varphi, \overline{\psi_2}, j)$
ssi $\forall 1 \leq j \leq i$, $u \in \mathcal{L}(\psi_2, j)$,
ou $\exists 1 \leq k \leq i$, $u \in \mathcal{L}(\psi_1, k)$ et $\forall k \leq j \leq i$, $u \in \mathcal{L}(\psi_2, j)$ (*h. d'induction*)
ssi $u \in \mathcal{L}(\psi_1 \widetilde{\mathsf{S}}\, \psi_2, i)$ (*sémantique de LTL*)

□

Nous pouvons maintenant aisément conclure : l'automate $\mathcal{A}_\varphi$ est bien celui que nous avions promis.

Théorème 8.15

Pour toute formule LTL φ, $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$.

Démonstration. Il suffit d'appliquer le lemme précédent à $\psi = \varphi$ et $i = 1$. □

8.4.3 Complexité

Pour toute formule PLTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U}\, \psi_2$, l'automate $\mathcal{A}_\varphi$ possède au maximum $n+1$ états et $n-r$ états répétés.

Chapitre 9

Automates de Büchi à mémoire

La construction classique pour transformer un automate alternant à double sens en automate de Büchi est complexe et coûteuse. Ici, nous allons pleinement utiliser le fait que nous partons d'un automate progressant très faible pour optimiser cette transformation. L'idée est la suivante : partant d'une exécution $\rho = (V, E, \sigma, \nu)$ d'un automate $\mathcal{A}$ sur un mot $u = u_1 u_2 \ldots \in \Sigma^\infty$, nous allons considérer les ensembles $X_i = \sigma(\nu^{-1}(i))$, $i \geq 0$, et construire un automate dont la suite $X_0, X_1, X_2, \ldots$ est une exécution, acceptante si et seulement si ρ est acceptante.

Pour cela, les transitions du nouvel automate doivent refléter le fait que chaque nœud x de ρ satisfait la fonction de transition de $\mathcal{A}$. Ainsi, étant donné que $\mathcal{A}$ est un automate à double sens, pour un nœud x d'étiquette de position i, il faut considérer les ensembles X_{i-1} et X_{i+1}. Il devient alors naturel de considérer des quadruplets $(X_{i-1}, X_i, X_{i+1}, u_i)$ comme transitions de notre nouvel automate. Ce nouveau type d'automate sera appelé *automate à mémoire*, cette terminologie étant expliquée un peu plus loin.

9.1 Définition

Définition 9.1 (*automate de Büchi à mémoire*)

Un automate de Büchi à mémoire est un quintuplet $\mathcal{M} = (Q, \Sigma, T, I, F, \mathcal{T})$ où :
- Q est l'ensemble (fini) des *états*,
- Σ est l'*alphabet*,
- $T \subseteq Q \times Q \times Q \times \Sigma$ est la fonction de *transition*,
- $I \subseteq Q \times Q$ est l'ensemble des *états initiaux*,
- $F \subseteq Q \times Q$ est l'ensemble des *états finaux*,
- $\mathcal{T} = \{T_1, \ldots, T_r\}$ est l'ensemble des *tables d'acceptation* où $\forall 1 \leq j \leq r$, $T_j \subseteq T$.

Soit $u = u_1 u_2 \ldots \in \Sigma^\infty$. Une *exécution* ρ de $\mathcal{M}$ sur u est une suite finie $q_0, q_1, \ldots, q_{|u|+1}$ si $|u| \in \mathbb{N}$, ou une suite infinie $q_0, q_1, \ldots$ si $|u| = \omega$, d'éléments de Q telle que :
- le premier état est initial : $(q_0, q_1) \in I$,
- on passe d'un état au suivant en suivant la fonction de transition :
  $\forall 1 \leq i \leq |u|$, $(q_{i-1}, q_i, q_{i+1}, u_i) \in T$.

Une exécution ρ est *acceptante* si u est fini et $(q_{|u|}, q_{|u|+1}) \in F$ ou si u est infini et ρ utilise une infinité de transitions $(q_{i-1}, q_i, q_{i+1}, u_i)$ dans chacune des tables T_j, $1 \leq j \leq r$.

Le *langage* $\mathcal{L}(\mathcal{M})$ d'un automate de Büchi à mémoire $\mathcal{M}$ est l'ensemble des mots de Σ^∞ sur lesquels il existe une exécution acceptante de $\mathcal{M}$.

Remarque.

Un automate de Büchi à mémoire peut être considéré comme un automate de Büchi généralisé dont l'ensemble d'états serait $Q \times Q$, et ayant la particularité que toute transition de (p_1, q_1) vers (p_2, q_2) vérifie $p_2 = q_1$. Ainsi on peut considérer que l'élément de droite de la paire est l'état courant, et que l'élément de gauche est l'état visité à l'étape précédente. Ainsi cet automate garde en mémoire le dernier état qu'il a visité, d'où le nom d'automate à mémoire.

9.2 Des automates progressants très faibles aux automates de Büchi à mémoire

9.2.1 Première construction

Dans cette section nous exposons comment, à partir d'un automate progressant très faible $\mathcal{A}$ avec n états, construire un automate de Büchi à mémoire $\mathcal{M}_\mathcal{A}$ de même langage, avec au plus 2^n états et n tables d'acceptation.

L'idée de cette construction est similaire au passage des automates alternant très faibles aux automates de Büchi généralisés de la section 5.3 : il s'agit de grouper les nœuds de même étiquette de position dans une exécution de $\mathcal{A}$, les états de $\mathcal{M}_\mathcal{A}$ devenant des sous-ensembles de l'ensemble des états de $\mathcal{A}$.

Comme pour le cas de LTL, la partie difficile de cette construction est de faire en sorte de pouvoir reconstruire, à partir d'une exécution acceptante de $\mathcal{M}_{\mathcal{A}}$, qui est une simple suite d'ensembles, une Q-forêt qui constitue une exécution acceptante de $\mathcal{A}$ sur le même mot. Pour pouvoir placer correctement les branches de cette forêt, nous aurons besoin des hypothèses que nous avons mentionnées précédemment, à savoir que $\mathcal{A}$ est très faible et progressant.

Définition 9.2 ($\mathcal{A} \to \mathcal{M}^1_{\mathcal{A}}$)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate progressant très faible. L'automate de Büchi à mémoire $\mathcal{M}^1_{\mathcal{A}} = (Q', \Sigma, T', I', F', \mathcal{T}')$ est défini comme suit :
- $Q' = 2^Q$,
- $T' = \{(\overleftarrow{X}, X, \overrightarrow{X}, a) \mid \exists(\overleftarrow{Y}, \overrightarrow{Y}) \in \bigotimes_{q \in X} \delta(q, a) \text{ avec } \overleftarrow{Y} \subseteq \overleftarrow{X} \text{ et } \overrightarrow{Y} \subseteq \overrightarrow{X}\}$,
- $I' = \{(X, Y) \in Q' \times Q' \mid X \subseteq F \text{ et } \exists Z \in I \text{ avec } Z \subseteq Y\}$,
- $F' = \{(X, Y) \in Q' \times Q' \mid Y \subseteq F\}$,
- $\mathcal{T}' = \{T'_q \mid q \in Q \backslash R\}$ où
  $T'_q = \{(\overleftarrow{X}, X, \overrightarrow{X}, a) \in T' \mid q \notin X \text{ ou } \exists(\overleftarrow{Y}, \overrightarrow{Y}) \in \delta(q, a), \overleftarrow{Y} \subseteq \overleftarrow{X}, \overrightarrow{Y} \subseteq \overrightarrow{X} \backslash \{q\}\}$.

9.2.2 Preuve de l'équivalence

Comme on l'attend, on a $\mathcal{L}(\mathcal{M}^1_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$. Cependant, nous ne prouverons ici qu'une seule des deux inclusions, l'autre inclusion étant une conséquence directe des résultats de la section qui suit. Voir le théorème 9.5 pour le résultat final.

Proposition 9.1

Pour tout automate progressant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{M}^1_{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$.

Démonstration. Soient $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{M}^1_{\mathcal{A}})$ et $X_0, X_1, \ldots$ une exécution acceptante de $\mathcal{M}^1_{\mathcal{A}}$ sur u. Nous allons construire un graphe étiqueté $\rho = (V, E, \sigma, \nu)$. Soit $V = \{(q, i) \mid i \geq 0,\ q \in X_i\}$. Par définition de T', $\forall i \in \mathbb{N}$, si $1 \leq i \leq |u|$, $\forall q \in X_i$, $\exists(\overleftarrow{Y}_{q,i}, \overrightarrow{Y}_{q,i}) \in \delta(q, u_i)$ avec $\overleftarrow{Y}_{q,i} \subseteq X_{i-1}$ et $\overrightarrow{Y}_{q,i} \subseteq X_{i+1}$, et si $q \notin R$ et $(X_{i-1}, X_i, X_{i+1}, u_i) \in T'_q$, on peut imposer en plus que $q \notin \overrightarrow{Y}_{q,i}$. Posons donc $E((q, i)) = \overleftarrow{Y}_{q,i} \times \{i - 1\} \cup \overrightarrow{Y}_{q,i} \times \{i + 1\}$. Si $i \in \{0, |u| + 1\}$ posons $E((q, i)) = \emptyset$. $\forall (q, i) \in V$, posons enfin $\sigma((q, i)) = q$ et $\nu((q, i)) = i$.

$(X_0, X_1) \in I'$ donc $\exists \Gamma \subseteq X_1 \times \{1\}$ tel que $\sigma(\Gamma) \in I$. Nous allons maintenant “déplier” le graphe ρ, à partir de l'ensemble de racines Γ. Nous obtenons de cette manière une Q-forêt $\rho' = (V', E', \sigma', \nu')$.

Soit $x \in V'$. Si $\nu'(x) \in \{0, |u| + 1\}$, alors $E'(x) = \emptyset$. Sinon, soient $q = \sigma'(x)$ et $i = \nu'(x)$, et posons $\overleftarrow{X} = \sigma'(\overleftarrow{E'}(x)) = \sigma(E(q, i) \cap Q \times \{i - 1\})$ et $\overrightarrow{X} = \sigma'(\overrightarrow{E'}(x)) = \sigma(E(q, i) \cap Q \times \{i + 1\})$. Par construction de ρ, $(\overleftarrow{X}, \overrightarrow{X}) \in \delta(q, u_i)$. ρ' est donc une exécution de $\mathcal{A}$ sur u.

Vérifions maintenant que ρ' est acceptante :

- $\sigma'(\nu'^{-1}(0)) \subseteq \sigma(\nu^{-1}(0)) = X_0 \subseteq F$, car $(X_0, X_1) \in I'$,
- si $|u| = n \in \mathbb{N}$ alors $\sigma'(\nu'^{-1}(n+1)) \subseteq \sigma(\nu^{-1}(n+1)) = X_{n+1} \subseteq F$, car $(X_n, X_{n+1}) \in F'$,
- Soit $x_0, x_1, \ldots$ une branche infinie de ρ'. $\mathcal{A}$ est très faible donc la suite des étiquettes d'état sur cette branche est ultimement constante : $\exists q \in Q$, $\exists N \geq 0$, $\forall i \geq N$, $\sigma(x_i) = q$. D'autre part $\mathcal{A}$ est progressant, donc la proposition 8.2 nous assure que ρ' est sans-boucle. Ainsi ν est strictement croissante à partir de la profondeur N. Supposons que $q \notin R$: $X_0, X_1, \ldots$ étant une exécution acceptante, $\exists k \geq \nu(x_N)$ tel que $(X_{k-1}, X_k, X_{k+1}, u_k) \in T'_q$. Il existe alors $j \geq N$ tel que $\lambda(x_j) = (q, k)$ et $\lambda(x_{j+1}) = (q, k+1)$. Mais lors de la construction de ρ, puisque $\sigma(x_j) = q$ et $(X_{k-1}, X_k, X_{k+1}, u_k) \in T'_q$, $(q, k+1) \notin E((q, k))$. Ceci contredit nos hypothèses, et donc ρ' est acceptante. Ainsi $u \in \mathcal{L}(\mathcal{A})$. □

9.2.3 Complexité

Pour tout automate progressant très faible $\mathcal{A}$ avec n états, l'automate de Büchi à mémoire $\mathcal{M}^1_{\mathcal{A}}$ possède au maximum 2^n états et n tables d'acceptation.

On remarque aisément que $\mathcal{M}^1_{\mathcal{A}}$ est toujours trop gros pour être utilisé dans une implémentation efficace. Il contient de nombreux états inutiles, et une restriction aux états accessibles serait insuffisante car les états initiaux sont déjà trop nombreux. En fait nous avons introduit $\mathcal{M}^1_{\mathcal{A}}$ pour pouvoir prouver que notre construction est correcte.

Le handicap de la méthode précédente est que, dans T', on accepte pour $\overleftarrow{X}$ et $\overrightarrow{X}$ tous les surensembles de $\overleftarrow{Y}$ et $\overrightarrow{Y}$. Pourquoi ne pas prendre uniquement $\overleftarrow{Y}$ et $\overrightarrow{Y}$ comme dans le cas de LTL ? C'est parce que l'automate que nous générons n'est pas à double sens, et qu'il doit "deviner" ce dont il aura besoin dans le futur. On doit donc nécessairement lui donner la capacité d'ajouter des éléments dans $\overleftarrow{Y}$ et $\overrightarrow{Y}$, chose qui était inutile pour LTL. Le problème est qu'autoriser tous les surensembles génère un nombre d'états bien trop grand.

9.2.4 Deuxième Construction

Si les exécutions de l'automate $\mathcal{M}^1_{\mathcal{A}}$ correspondaient à la mise à plat des exécutions de $\mathcal{A}$, celles de notre nouvel automate $\mathcal{M}^2_{\mathcal{A}}$ correspondront à la mise à plat des exécutions *minimales* de $\mathcal{A}$, c'est-à-dire pour lesquelles la suppression de tout sous-arbre rend l'exécution invalide.

L'intuition de la deuxième construction est la suivante : puisque l'automate ne peut pas revenir dans le passé au cours d'une exécution, nous allons revenir dans le passé au cours de sa construction. Nous partirons avec des ensembles minimaux, mais si un élément manque, alors nous allons créer des nouveaux états qui en tiennent compte, et explorer les états devenus accessibles par cet ajout.

Au cours de cette construction, les états de l'automate de Büchi à mémoire seront stockés sous la forme de paires $(X,Y) \in Q \times Q$, Y représentant l'état courant, et X l'état précédent. Si on se rend compte que X n'est pas assez gros pour satisfaire les conditions imposées par Y dans δ, la construction reviendra en arrière en agrandissant l'état X.

Définition 9.3 ($\mathcal{A} \to \mathcal{M}^2_{\mathcal{A}}$)

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate progressant très faible.
L'automate de Büchi à mémoire $\mathcal{M}^2_{\mathcal{A}} = (Q'', \Sigma, T'', I'', F'', \mathcal{T}'')$ est construit ainsi :

Initialisation :
$I'' = \{F\} \times I, \quad \nabla = \{F\} \times I, \quad T'' = \emptyset.$

Saturation : nous appliquons la procédure de saturation suivante à tous les éléments (X,Y) de ∇, jusqu'à obtenir un point fixe pour I'', ∇, et T''.

```
    for each a ∈ Σ, for each (X', Z) ∈ ⊗_{q∈Y} δ(q, a),
(a)  if X' ⊆ X
       add (Y, Z) to ∇
       add (X, Y, Z, a) to T''
     else
(b)    for each (W, X, Y, b) ∈ T'' with (W, X) ∈ I''
         add (W, X ∪ X') to ∇
         add (W, X ∪ X') to I''
(c)    for each (V, W, X, c), (W, X, Y, b) ∈ T''
         add (W, X ∪ X') to ∇
         add (V, W, X ∪ X', c) to T''
```

Finalisation :
$Q'' = \{X \in 2^Q \mid \exists Y \in 2^Q, (X,Y) \in \nabla \text{ ou } (Y,X) \in \nabla\}$,
$F'' = \{(X,Y) \in \nabla \mid Y \subseteq F\}$,
$\mathcal{T}'' = \{T''_q \mid q \in Q \backslash R\}$ où
$T''_q = \{(\overleftarrow{X}, X, \overrightarrow{X}, a) \in T'' \mid q \notin X \text{ ou } \exists(\overleftarrow{Y}, \overrightarrow{Y}) \in \delta(q,a),\ \overleftarrow{Y} \subseteq \overleftarrow{X},\ \overrightarrow{Y} \subseteq \overrightarrow{X} \backslash \{q\}\}$.

9.2.5 Preuve de l'équivalence

Nous allons prouver deux résultats qui, combinés à la proposition 9.1, permettront d'obtenir le résultat attendu.

Proposition 9.2

Pour tout automate progressant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{M}^2_{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{M}^1_{\mathcal{A}})$.

Démonstration. On note aisément que tout état de $\mathcal{M}^2_{\mathcal{A}}$ apparaît aussi dans $\mathcal{M}^1_{\mathcal{A}}$. En ce qui concerne les transitions, on peut montrer par induction qu'au fur et à mesure de la construction on a toujours $T'' \subseteq T'$. À l'initialisation, $T'' = \emptyset$. Dans le cas (a), $(X', Z) \in \bigotimes_{q \in Y} \delta(q, a)$ et $X' \subseteq X$ donc $(X, Y, Z, a) \in T'$, et dans le cas (c), $(V, W, X, c) \in T'' \subseteq T'$ donc $\exists(\overleftarrow{Y}, \overrightarrow{Y}) \in \bigotimes_{q \in W} \delta(q, c)$ avec $\overleftarrow{Y} \subseteq V$ et $\overrightarrow{Y} \subseteq X \subseteq X \cup X'$ donc $(V, W, X \cup X', c) \in T'$.

Ainsi toute exécution acceptante de $\mathcal{M}^2_{\mathcal{A}}$ sur un mot u est aussi une exécution acceptante de $\mathcal{M}^1_{\mathcal{A}}$ sur le même mot. □

Proposition 9.3

Pour tout automate progressant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{M}^2_{\mathcal{A}})$.

Démonstration. Soient $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$ et $\rho = (V, E, \sigma, \nu)$ une exécution acceptante de $\mathcal{A}$ sur u. Nous allons construire par récurrence sur k une suite $\rho'_k = (X_i)_{0 \leq i \leq n}$ telle que :

- $1 \leq n \leq |u| + 1$,
- $(X_0, X_1) \in I''$,
- $\forall 1 \leq i \leq n$, $X_i \subseteq \sigma(\nu^{-1}(i))$ et $(X_{i-1}, X_i) \in \nabla$,
- $\forall 1 \leq i < n$, $(X_{i-1}, X_i, X_{i+1}, u_i) \in T''$,
- $\forall 1 \leq i < n$, $\forall q \in X_i \backslash R$, si il existe $x \in \nu^{-1}(i)$ tel que $\sigma(x) = q$ et $q \notin \sigma(\overrightarrow{E}(x))$ alors $(X_{i-1}, X_i, X_{i+1}, u_i) \in T''_q$.

Posons pour commencer $\rho'_0 = X_0, X_1$ avec $X_0 = F$ et $X_1 = \sigma(\Gamma)$. $X_1 \in I$ donc $(X_0, X_1) \in I'' \subseteq \nabla$. L'hypothèse de récurrence est vérifiée.

Supposons maintenant que nous avons construit une suite $\rho'_k = (X_i)_{0 \leq i \leq n}$ avec $1 \leq n \leq |u|$, qui satisfait l'hypothèse de récurrence. Si $n = |u| + 1$ nous stoppons ici la construction.

Sinon, $\forall q \in X_n$, choisissons $x_q \in V$ tel que $\lambda(x_q) = (q, n)$, et tel que si $q \notin R$ alors $q \notin \sigma(\overrightarrow{E}(x_q))$ si c'est possible. Soient $X = \{x_q \mid q \in X_n\}$, $X'_{n-1} = \sigma(\overleftarrow{E}(X))$, $X_{n+1} = \sigma(\overrightarrow{E}(X))$. Comme ρ est une exécution de $\mathcal{A}$ sur u et comme on a $n \leq |u|$, $(X'_{n-1}, X_{n+1}) \in \bigotimes_{q \in X_n} \delta(q, u_n)$.

Comme ρ'_k satisfait l'hypothèse de récurrence, on a $(X_{n-1}, X_n) \in \nabla$ et trois cas sont possibles :

(a) si $X'_{n-1} \subseteq X_{n-1}$, alors d'après la construction de $\mathcal{M}^2_{\mathcal{A}}$, on a $(X_n, X_{n+1}) \in \nabla$ et $(X_{n-1}, X_n, X_{n+1}, u_n) \in T''$. De plus, $\forall q \in X_n \setminus R$, si $\exists x \in \nu^{-1}(n)$ avec $\sigma(x) = q$ et $q \notin \sigma(\overrightarrow{E}(x))$, alors $\exists X' \subseteq X_{n-1}$, $\exists Z' \subseteq X_{n+1} \backslash \{q\}$, $(X', Z') \in \delta(q, a)$ et ainsi $(X_{n-1}, X_n, X_{n+1}, u_n) \in T''_q$. Par conséquent $\rho'_{k+1} = (X_i)_{0 \leq i \leq n+1}$ satisfait l'hypothèse de récurrence.

Supposons maintenant que $X'_{n-1} \not\subseteq X_{n-1}$. Notons que ceci implique $n \geq 2$, car si $n = 1$ alors, ρ étant acceptant, $X'_0 \subseteq F = X_0$.

(b) Si $n = 2$ alors $(X_0, X_1, X_2, u_1) \in T''$ et $(X_0, X_1) \in I''$ donc d'après la construction de $\mathcal{M}^2_{\mathcal{A}}$, $(X_0, X_1 \cup X'_1) \in I'' \subseteq \nabla$. Ainsi $\rho'_{k+1} = X_0, X_1 \cup X'_1$ satisfait l'hypothèse de récurrence.

(c) Supposons maintenant que $n \geq 3$. Alors on a $(X_{n-3}, X_{n-2}, X_{n-1}, u_{n-2}) \in T''$ et $(X_{n-2}, X_{n-1}, X_n, u_{n-1}) \in T''$. D'après la construction de $\mathcal{M}^2_{\mathcal{A}}$, on a nécessairement $(X_{n-3}, X_{n-2}, X_{n-1} \cup X'_{n-1}, u_{n-2}) \in T''$ et $(X_{n-2}, X_{n-1} \cup X'_{n-1}) \in \nabla$. De plus, si $(X_{n-3}, X_{n-2}, X_{n-1}, u_{n-2}) \in T''_q$ alors $(X_{n-3}, X_{n-2}, X_{n-1} \cup X'_{n-1}, u_{n-2}) \in T''_q$. Par conséquent $\rho'_{k+1} = X_0, X_1, \dots, X_{n-2}, X_{n-1} \cup X'_{n-1}$ satisfait l'hypothèse de récurrence.

Nous allons maintenant prouver que la suite $(\rho'_k)_{k\geq 0}$ converge. Pour cela, considérons l'alphabet fini $A = 2^Q$ partiellement ordonné par l'inclusion. La suite de mots $(\rho'_k)_{k\geq 0}$ sur l'alphabet A est strictement croissante pour l'ordre lexicographique induit par l'inclusion sur A^*. Par conséquent soit cette suite est finie, soit elle est infinie et converge vers un mot infini $\rho' \in A^\omega$ (voir le lemme 9.4 ci-dessous).

Si u est fini alors le mots ρ'_k sont de longueur au plus $|u|+2$. Ainsi la construction des ρ_k termine. Soit ρ' le dernier ρ'_k : ρ' est une exécution de $\mathcal{M}^2_{\mathcal{A}}$ sur u. Comme ρ est acceptante, $\sigma(\nu^{-1}(|u|+1)) \subseteq F$ donc $(X_{|u|}, X_{|u|+1}) \in F''$. Ainsi ρ' est acceptante, et $u \in \mathcal{L}(\mathcal{M}^2_{\mathcal{A}})$.

Si u est infini, alors la suite $(\rho'_k)_{k\geq 0}$ est infinie et converge vers un mot infini $\rho' \in A^\omega$. Tout préfixe de ρ' étant le préfixe d'un des ρ_k, ρ' est une exécution de $\mathcal{M}^2_{\mathcal{A}}$ sur u. Supposons que ρ' n'est pas acceptante : il existe $q \in Q \backslash R$ tel que après un rang N, aucune transition de ρ' n'est dans T''_q. Ainsi $\forall x \in V$ si $\lambda(x) = (q, i)$ avec $i \geq N$ alors $q \in \sigma(\overrightarrow{E}(x))$. De plus, si $(X_{N-1}, X_N, X_{N+1}, u_N) \notin T''_q$, alors $q \in X_N$. Ainsi il existe dans ρ une branche infinie dont les étiquettes d'état sont ultimement égales à q, ce qui contredit l'hypothèse que ρ est acceptante. Ainsi $u \in \mathcal{L}(\mathcal{M}^2_{\mathcal{A}})$. □

Lemme 9.4

Soit A un ensemble fini, partiellement ordonné par $\preccurlyeq$, et soit $w_0, w_1, \dots$ une suite infinie de mots de A^*. Si $(w_i)_{i\geq 0}$ est strictement croissante pour l'ordre lexicographique induit par $\preccurlyeq$, alors cette suite converge vers une limite $w \in A^\omega$.

Démonstration. $\forall i \geq 0$, notons $w_i = u_i^1 \dots u_i^{n_i}$. $(w_i)_{i\geq 0}$ est strictement croissante pour l'ordre lexicographique induit par $\preccurlyeq$, donc $(u_i^1)_{i>0}$ est croissante pour $\preccurlyeq$. Comme A est fini, cette suite atteint une limite u^1 pour un certain $i = i^1$. Pour $i \geq i^1$, $u_i^1 = u^1$ donc $(u_i^2)_{i>i^1}$ est croissante, et atteint une limite u^2 pour $i = i^2$. En continuant de cette manière, on obtient un mot $w = u^1 u^2 \dots \in A^\omega$, limite de la suite $(w_i)_{i\geq 0}$. □

Les propositions 9.1, 9.2 et 9.3 permettent de conclure sur le résultat attendu :

Théorème 9.5

Pour tout automate progressant très faible $\mathcal{A}$, $\mathcal{L}(\mathcal{M}^1_{\mathcal{A}}) = \mathcal{L}(\mathcal{M}^2_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$.

9.2.6 Complexité

Pour tout automate progressant très faible $\mathcal{A}$ avec n états et $n - r$ états répétés, l'automate de Büchi à mémoire $\mathcal{M}^2_{\mathcal{A}}$, comme $\mathcal{M}^1_{\mathcal{A}}$, possède au maximum 2^n états et r tables d'acceptation. Cependant $\mathcal{M}^2_{\mathcal{A}}$ est toujours plus petit que $\mathcal{M}^1_{\mathcal{A}}$, et en pratique il possède beaucoup moins d'états et de transitions, comme le montrent les expérimentations de la section 11.4.

Ainsi pour toute formule PLTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U} \psi_2$, nous sommes capables de générer un automate de Büchi à mémoire de même langage, avec au maximum 2^{n+1} états et r tables d'acceptation.

Chapitre 10

Model-checking pour PLTL

Nous savons maintenant générer à partir de toute formule PLTL φ de taille temporelle n un automate de Büchi à mémoire avec au maximum 2^{n+1} états et n tables d'acceptation, reconnaissant le langage de φ.

Comme dans le cadre de LTL, nous allons proposer deux approches : convertir l'automate de Büchi à mémoire en un automate de Büchi classique pour pouvoir utiliser des algorithmes de model-checking existants, ou adapter les procédures classiques de model-checking aux automates de Büchi à mémoire.

10.1 Des automates de Büchi à mémoire aux automates de Büchi classiques

Dans cette section nous allons voir comment, à partir d'un automate de Büchi à mémoire $\mathcal{M}$ avec n états et r tables d'acceptation, construire un automate de Büchi classique $\mathcal{B}_{\mathcal{M}}$ avec au plus $n^2 \times (r+1)$ états.

10.1.1 Construction

L'idée de cette construction est simple : il suffit de transformer l'automate de Büchi à mémoire $\mathcal{M}$ en un automate de Büchi généralisé $\mathcal{G}_{\mathcal{M}}$ sur $Q \times Q$, et d'appliquer ensuite la procédure exposée dans la section 6.1, pour obtenir un automate de Büchi $\mathcal{B}_{\mathcal{M}}$.

On ajoutera cependant un élément dans la construction de $\mathcal{B}_{\mathcal{M}}$: on définira pour l'acceptation des mots finis un ensemble $F' = F \times \{0, \ldots, r\}$.

Définition 10.1 ($\mathcal{M} \to \mathcal{G}_{\mathcal{M}}$)

Soit $\mathcal{M} = (Q, \Sigma, T, I, F, \mathcal{T})$ un automate de Büchi à mémoire, avec $\mathcal{T} = \{T_1, \ldots, T_r\}$. L'automate de Büchi généralisé $\mathcal{G}_{\mathcal{M}} = (Q', \Sigma, T', I', F', \mathcal{T}')$ est défini comme suit :

- $Q' = Q \times Q$,
- $T' = \{((\overleftarrow{q}, q), a, (q, \overrightarrow{q})) \mid (\overleftarrow{q}, q, \overrightarrow{q}, a) \in T\}$,
- $I' = I$,
- $F' = F$,
- $\mathcal{T}' = \{T'_1, \ldots, T'_r\}$ où $\forall 1 \leq j \leq r$, $T'_j = \{((\overleftarrow{q}, q), a, (q, \overrightarrow{q})) \mid (\overleftarrow{q}, q, \overrightarrow{q}, a) \in T_j\}$.

Il est clair que l'automate $\mathcal{G}_{\mathcal{M}}$ possède le même langage que $\mathcal{M}$.

10.1.2 Complexité

Pour tout automate de Büchi à mémoire $\mathcal{M}$ avec n états et r conditions d'acceptation, l'automate $\mathcal{G}_{\mathcal{M}}$ possède n^2 états et r conditions d'acceptation, et l'automate $\mathcal{B}_{\mathcal{M}}$ possède au maximum $n^2 \times (r+1)$ états.

Ainsi pour toute formule PLTL φ de taille temporelle n avec r sous-formules du type $\psi_1 \mathsf{U} \psi_2$, nous sommes capables de générer un automate de Büchi de même langage, avec au maximum $(r+1) \times 2^{2n+2}$ états.

À titre de comparaison, dans [KPV01], Kupferman et al., à partir d'un formule ETL_{2a} de taille temporelle n, génèrent un automate de Büchi de même langage avec au maximum $2^{\mathcal{O}(n^2)}$ états. ETL_{2a} est une logique temporelle qui inclut PLTL. Nous sommes donc capables, sur une logique plus restreinte que ETL_{2a}, de construire un automate de taille maximale inférieure.

10.2 Automates de Büchi à mémoire et model-checking

Comme pour LTL, il est intéressant d'effectuer le model-checking directement à partir d'un automate de Büchi à mémoire, en calculant le produit de cet automate avec le modèle à vérifier. Il suffit alors de calculer à l'aide de l'algorithme de Tarjan les composantes fortement connexes du produit, et d'effectuer le test du vide.

La construction est la même que dans le cas de LTL, il suffit ici encore de considérer l'automate de Büchi à mémoire $\mathcal{M}$ comme un automate de Büchi généralisé $\mathcal{G}_{\mathcal{M}}$ sur $Q \times Q$, et d'appliquer ensuite la procédure exposée dans la section 6.2.

Le temps de calcul pour le test du vide sur un automate de Büchi généralisé est linéaire en la taille de l'automate $\mathcal{G}_{\mathcal{M}}$. Or si $\mathcal{M}$ possède n états et r tables d'acceptations, la taille de $\mathcal{G}_{\mathcal{M}}$ et celle de $\mathcal{M}$ sont toutes les deux $\mathcal{O}(n^3 \times r \times |\Sigma|)$, car toutes les transitions de $Q^2 \times \Sigma \times Q^2$ dans $\mathcal{G}_{\mathcal{M}}$ sont de la forme $((\overleftarrow{q}, q), a, (q, \overrightarrow{q}))$, et donc la taille de $\mathcal{T}'$ est $\mathcal{O}(n^3 \times r \times |\Sigma|)$.

Ainsi le temps de calcul pour le test du vide sur un automate de Büchi à mémoire reste linéaire en la taille de l'automate.

Chapitre 11

Implémentation

Nous allons évoquer ici les optimisations qui peuvent être effectuées pour améliorer l'algorithme présenté au cours cette partie, ainsi que quelques résultats expérimentaux de notre implémentation.

11.1 Optimisation des structures de données

Comme pour LTL, nous avons optimisé nos structures de données pour l'implémentation, d'une part pour réduire l'espace mémoire occupé et améliorer ainsi les performances de notre algorithme et pouvoir transformer des formules PLTL qu'il eut été trop coûteux de traiter sans ces optimisations, et d'autre part pour simplifier les calculs à effectuer, ce qui permet de réduire le temps de calcul nécessaire et donc d'améliorer encore nos performances.

11.1.1 Étiquetage des transitions

Nous avons choisi de réutiliser nos optimisations sur l'étiquetage des transitions, comme pour PLTL, en représentant les éléments de 2^{Σ} par deux mots de $|\mathrm{AP}|$ bits, d'une part pour réduire l'espace mémoire occupé (il fallait $2^{|\mathrm{AP}|}$ bits avec la représentation de base), et d'autre part pour simplifier les calculs d'intersection et d'inclusion.

Pour plus de détails sur l'optimisation des étiquettes des transitions, voir la section 7.1.1.

11.1.2 Transitions des automates alternants

Pour LTL, dans la section 7.1.2, nous avons proposé une nouvelle représentation de la fonction de transition des automates alternants, afin de réduire le nombre de transitions et d'améliorer la conversion en un automate de Büchi généralisé.

Pour PLTL, nous avons en fait choisi d'utiliser directement la version optimisée, et c'est celle-ci que nous avons manipulée tout au long de la partie. Ceci nous a permis de

mieux faire comprendre comment manipuler ce formalisme, moins intuitif que l'usage des formules de $\mathcal{B}^+(Q \times \{-1, 1\})$.

D'autre part, seule cette définition des automates alternants permet d'utiliser l'algorithme de construction de l'automate $\mathcal{M}^2_{\mathcal{A}}$, présentée dans la section 9.2. Avec la définition "classique", cet algorithme serait loin d'avoir la même efficacité.

11.2 Simplification des automates

Comme dans le cadre de LTL abordé dans la section 7.2, nous avons la capacité d'améliorer grandement les performances de notre algorithme en simplifiant chaque automate avant de passer à l'étape suivante.

Les simplifications sont les mêmes que pour LTL : simplification des transitions, simplification des états, et utilisation des composantes fortement connexes.

11.2.1 Simplification des transitions

De même que pour LTL, si une transition t_2 est plus contraignante qu'une transition t_1, alors la transition t_2 peut être supprimée sans changer le langage de l'automate. La notion "être plus contraignante que" dépend de l'automate concerné.

Proposition 11.1 (*automate alternant*)

> *Soit $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ un automate alternant très faible à double sens (défini sous sa version optimisée). Si $\exists q \in Q\ \exists(\overleftarrow{X_1}, \alpha_1, \overrightarrow{X_1})$, $(\overleftarrow{X_2}, \alpha_2, \overrightarrow{X_2}) \in \delta(q)$ tels que $\overleftarrow{X_1} \subseteq \overleftarrow{X_2}$, $\overrightarrow{X_1} \subseteq \overrightarrow{X_2}$ et $\alpha_2 \subseteq \alpha_1$, mais $(\overleftarrow{X_1}, \alpha_1, \overrightarrow{X_1}) \neq (\overleftarrow{X_2}, \alpha_2, \overrightarrow{X_2})$, alors on peut supprimer $(\overleftarrow{X_2}, \alpha_2, \overrightarrow{X_2})$ de $\delta(q)$.*

Démonstration. Soit $\mathcal{A}'$ l'automate modifié, après suppression de la transition. Nous allons prouver que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

$\boxed{\subseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$ et soit $\rho = (V, E, \sigma, \nu)$ une exécution acceptante de $\mathcal{A}$ sur u. Pour tout nœud $x \in V$ d'étiquette q tel que $u_{\nu(x)} \in \alpha_2$ et $(\sigma(\overleftarrow{E}(x)), \sigma(\overrightarrow{E}(x))) = (\overleftarrow{X_2}, \overrightarrow{X_2})$, supprimons tous les fils gauches de x d'étiquette $p \in \overleftarrow{X_2} \backslash \overleftarrow{X_1}$, tous les fils droits de x d'étiquette $p \in \overrightarrow{X_2} \backslash \overrightarrow{X_1}$ et leurs sous-arbres. On a alors $(\sigma(\overleftarrow{E}(x)), \sigma(\overrightarrow{E}(x))) = (\overleftarrow{X_1}, \overrightarrow{X_1})$ et $u_{\nu(x)} \in \alpha_1$ car $\alpha_2 \subseteq \alpha_1$. La Q-forêt ρ' ainsi obtenue reste une exécution acceptante de $\mathcal{A}$ sur u et n'utilise plus la transition $(\overleftarrow{X_2}, q, \overrightarrow{X_2}, \alpha_2)$: c'est une exécution acceptante de $\mathcal{A}'$ sur u, et $u \in \mathcal{L}(\mathcal{A}')$.

$\boxed{\supseteq}$ Toute exécution acceptante de l'automate $\mathcal{A}'$ sur un mot u est aussi une exécution acceptante de $\mathcal{A}$ sur u. $\square$

Proposition 11.2 (*automate de Büchi à mémoire*)

Soit $\mathcal{M} = (Q, \Sigma, T, I, F, \mathcal{T})$ un automate de Büchi à mémoire (défini sous sa version optimisée). Si $\exists(\overleftarrow{q}, q, \overrightarrow{q}, \alpha_1)$, $(\overleftarrow{q}, q, \overrightarrow{q}, \alpha_2) \in T$ tels que $\alpha_2 \subsetneq \alpha_1$ et $\forall T_j \in \mathcal{T}$, $(\overleftarrow{q}, q, \overrightarrow{q}, \alpha_2) \in T_j \Rightarrow (\overleftarrow{q}, q, \overrightarrow{q}, \alpha_1) \in T_j$, alors on peut supprimer $(\overleftarrow{q}, q, \overrightarrow{q}, \alpha_2)$ de T.

Démonstration. Ce résultat est clair car si $u_i \in \alpha_2$ alors $u_i \in \alpha_1$, et ainsi toute exécution acceptante de $\mathcal{M}$ peut être considérée comme une exécution acceptante de l'automate modifié et réciproquement. □

Proposition 11.3 (*automate de Büchi*)

Soit $\mathcal{B} = (Q, \Sigma, \delta, I, F, R)$ un automate de Büchi (défini sous sa version optimisée). Si $\exists q \in Q \ \exists(\alpha_1, q'), (\alpha_2, q') \in \delta(q)$ tels que $\alpha_2 \subsetneq \alpha_1$, alors on peut supprimer (α_2, q') de $\delta(q)$.

Démonstration. Voir la démonstration de la proposition 11.2 □

11.2.2 Simplification des états

Si deux états d'un automate sont équivalents alors ils peuvent être fusionnés, sans changer le langage de l'automate concerné. La notion d'états "équivalents" dépend de l'automate concerné.

- Pour un automate alternant très faible à double sens, q_1 et q_2 sont équivalents si $\delta(q_1) = \delta(q_2)$, $q_1 \in F \iff q_2 \in F$ et $q_1 \in R \iff q_2 \in R$.
- Pour un automate de Büchi à mémoire, q_1, q_2 sont équivalents si $q_1 \in F \iff q_2 \in F$, $\forall(\overleftarrow{q}, \overrightarrow{q}, \alpha) \in Q \times Q \times 2^{\Sigma}$, $(\overleftarrow{q}, q_1, \overrightarrow{q}, \alpha) \in T \iff (\overleftarrow{q}, q_2, \overrightarrow{q}, \alpha) \in T$, et $\forall T_j \in \mathcal{T}$, $(\overleftarrow{q}, q_1, \overrightarrow{q}, \alpha) \in T_j \iff (\overleftarrow{q}, q_2, \overrightarrow{q}, \alpha) \in T_j$.
- Pour un automate de Büchi, q_1 et q_2 sont équivalents si $\delta(q_1) = \delta(q_2)$, $q_1 \in F \iff q_2 \in F$ et $q_1 \in R \iff q_2 \in R$.

La fusion de deux états consiste à supprimer un des états, et remplacer dans la fonction de transition et dans la condition initiale toute occurence de cet état par une occurence de l'autre état.

Nous allons prouver que ces règles de simplification ne changent pas le langage de l'automate. Seul le cas de l'automate alternant sera traité, les autres cas pouvant être prouvés avec la même démonstration

Proposition 11.4

Soit $\mathcal{A} = (Q, \Sigma, \delta, I, F, R)$ un automate alternant très faible à double sens (défini sous sa version optimisée). Si $\exists q_1, q_2 \in Q$, $q_1 \neq q_2$ tels que $\delta(q_1) = \delta(q_2)$, $q_1 \in F \iff q_2 \in F$ et $q_1 \in R \iff q_2 \in R$ alors on peut supprimer de Q l'état q_2, en remplaçant chaque occurence de q_2 dans I et dans δ par q_1.

Démonstration. Soit $\mathcal{A}'$ l'automate modifié, après fusion des états.
Nous allons prouver que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

$\boxed{\subseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A})$ et soit ρ une Q-forêt, exécution acceptante de $\mathcal{A}$ sur u. En remplaçant dans ρ toutes les étiquettes q_2 par des étiquettes q_1, on obtient clairement une exécution acceptante de $\mathcal{A}'$ sur u, et $u \in \mathcal{L}(\mathcal{A}')$.

$\boxed{\supseteq}$ Soit $u = u_1 u_2 \ldots \in \mathcal{L}(\mathcal{A}')$ et soit $\rho = (V, E, \sigma)$ une exécution acceptante de $\mathcal{A}'$ sur u. Soient $y \in V$, $(q, i) = \lambda(y)$, $\overleftarrow{X'} = \sigma(\overleftarrow{E}(y))$, $\overrightarrow{X'} = \sigma(\overrightarrow{E}(y))$. Si $1 \leq i \leq |u|$, alors $\exists \alpha \in 2^{\Sigma}$ tel que $u_i \in \alpha$ et $(\overleftarrow{X'}, \alpha, \overrightarrow{X'}) \in \delta'(q)$. Il existe $(\overleftarrow{X}, \alpha, \overrightarrow{X}) \in \delta(q)$ tel que le remplacement de q_2 par q_1 dans $\overleftarrow{X}$ et $\overrightarrow{X}$ produit $\overleftarrow{X'}$ et $\overrightarrow{X'}$.
- si $q_2 \notin \overleftarrow{X}$, on ne change rien,
- si $q_2 \in \overleftarrow{X}$ et $q_1 \notin \overleftarrow{X}$, on change σ pour que les fils gauches de x d'étiquette q_1 aient pour nouvelle étiquette q_2,
- si $q_2 \in \overleftarrow{X}$ et $q_1 \in \overleftarrow{X}$, on duplique un fils gauche de x d'étiquette q_1 et son sous-arbre, et la copie de ce fils prend pour étiquette q_2.

Après avoir effectué les mêmes opérations pour les fils droits de x, $\sigma(\overleftarrow{E}(y)) = \overleftarrow{X}$ et $\sigma(\overrightarrow{E}(y)) = \overrightarrow{X}$.

De même, pour les racines, $\sigma(\Gamma) \in I'$ donc il existe $X \in I$ tel que le remplacement de q_2 par q_1 dans X produit $\sigma(\Gamma)$. À nouveau, trois cas sont possibles :
- si $q_2 \notin X$, on ne change rien,
- si $q_2 \in X$ et $q_1 \notin X$, on change σ pour que les racines d'étiquette q_1 aient pour nouvelle étiquette q_2,
- si $q_2 \in X$ et $q_1 \in X$, on duplique une racine d'étiquette q_1 et son sous-arbre, et la copie de cette racine prend pour étiquette q_2.

On se convaincra aisément que la Q-forêt ainsi obtenue devient une exécution acceptante de $\mathcal{A}$ sur u, et $u \in \mathcal{L}(\mathcal{A})$. □

Dans le cas d'un automate de Büchi à mémoire, il existe cependant une méthode plus efficace pour simplifier les états : il faut transformer l'automate $\mathcal{M}$ en un automate de Büchi généralisé $\mathcal{G}_{\mathcal{M}}$ comme exposé dans la section 10.1.1. Cet automate peut alors être simplifié par la méthode détaillée dans la section 7.2.2. Cependant, une fois simplifié, cet automate n'est plus un automate de Büchi à mémoire. Le model-checking ou la conversion en automate de Büchi se font alors comme dans le cas de LTL.

11.3 Construction de l'automate de Büchi à mémoire

Pour générer toutes les transitions de l'automate $\mathcal{M}^2_{\mathcal{A}}$, il faut répéter la procédure de saturation jusqu'à ce que plus aucun changement n'intervienne. Cette procédure doit être implémentée avec précaution pour éviter les calculs redondants. La méthode que nous avons retenue consiste à dater tous les états et les transitions par des entiers. Voici la nouvelle procédure de saturation, elle est détaillée dans la figure 11.1 et expliquée ci-dessous.

```
    increase current_timestamp
    for each a ∈ Σ, for each (X', Z) ∈ ⊗_{q∈Y} δ(q, a),
    ⎧ if X' ⊆ X
    ⎪   if (X, Y).timestamp = 0
    ⎪     if (Y, Z) ∉ ∇
(a) ⎨       add (Y, Z) to ∇
    ⎪       let (Y, Z).timestamp = 0
    ⎪     if (X, Y, Z, a) ∉ T''
    ⎪       add (X, Y, Z, a) to T''
    ⎩       let (X, Y, Z, a).timestamp = current_timestamp
     else
    ⎧   for each (W, X, Y, b) ∈ T'' with (W, X) ∈ I''
    ⎪     if (W, X, Y, b).timestamp ≥ (X, Y).timestamp
(b) ⎨       if (W, X ∪ X') ∉ ∇
    ⎪         add (W, X ∪ X') to ∇
    ⎪         let (W, X ∪ X').timestamp = 0
    ⎩       add (W, X ∪ X') to I''
    ⎧   for each (V, W, X, c), (W, X, Y, b) ∈ T''
    ⎪     if (V, W, X, c).timestamp ≥ (X, Y).timestamp
    ⎪     or (W, X, Y, b).timestamp ≥ (X, Y).timestamp
    ⎪       if (W, X ∪ X') ∉ ∇
(c) ⎨         add (W, X ∪ X') to ∇
    ⎪         let (W, X ∪ X').timestamp = 0
    ⎪       if (V, W, X ∪ X', c) ∉ T''
    ⎪         add (V, W, X ∪ X', c) to T''
    ⎩         let (V, W, X ∪ X', c).timestamp = current_timestamp

    let (X, Y).timestamp = current_timestamp
```

TAB. 11.1 – Procédure de saturation optimisée pour la construction de $\mathcal{M}^2_{\mathcal{A}}$

Lorsqu'un élément de ∇ est créé, il est daté à 0, alors que les nouvelles transitions sont datées à la date courante. A chaque fois qu'une paire (X, Y) est traitée dans la procédure de saturation, la date courante est incrémentée, et une fois que la paire a été traitée, elle est redatée à la date courante.

Pour éviter les calculs redondants, des gardes sont ajoutées pour chaque étape :

- l'étape `(a)` est exécutée uniquement si la date de (X, Y) est nulle. Ainsi, cette étape n'est exécutée qu'une seule fois, car il est clair que la création de nouveaux états ou transitions n'influe pas sur le résultat.
- dans l'étape `(b)`, on considère uniquement les transitions (W, X, Y, b) datées à une date supérieure ou égale à la date de (X, Y). Ainsi on ne retraite cette étape qu'avec

les nouvelles transitions ajoutées à T'' depuis le dernier passage.

– de même, dans l'étape `(c)`, on ne considère uniquement les paires de transitions pour lesquelles au moins une des transitions est datée à une date supérieure ou égale à la date de (X, Y). Ainsi, encore une fois, une paire de transitions déjà traitée ne sera pas inutilement retraitée.

Le nouvel algorithme que nous proposons ici permet donc d'obtenir le même automate que l'algorithme précédent, mais il évite d'effectuer inutilement les mêmes calculs à plusieurs reprises.

11.4 Résultats expérimentaux

Nous avons pour finir réalisé une implémentation de notre algorithme. Pour PLTL, nous n'avons pas trouvé d'autre implémentation disponible. Nous avons donc simplement effectué quelques comparaisons entre les algorithmes qui génèrent $\mathcal{M}^1_{\mathcal{A}}$ et $\mathcal{M}^2_{\mathcal{A}}$, afin de prouver l'intérêt de la deuxième construction.

Les calculs ont été effectués sur les formules

$$\pi_n = \neg\, \mathsf{G}(p_1 \rightarrow \mathsf{O}(p_2 \wedge \mathsf{O}(p_3 \wedge \ldots \mathsf{O}\, p_n) \ldots)$$

qui expriment que chaque p_1 est précédé d'un p_2, lui-même précéde de p_3, et ainsi de suite jusqu'à p_n.

	$\mathcal{A}_\pi$	$\mathcal{M}^1_{\mathcal{A}}$				$\mathcal{M}^2_{\mathcal{A}}$			
	états	avant	après	temps	mémoire	avant	après	temps	mémoire
π_2	3	28, 82	6, 11	0,03	<380	7,10	4, 6	0,01	<380
π_3	4	100, 544	12, 36	0,83	<380	10,19	6,13	0,01	<380
π_4	5	364, 3630	27,102	230	1 700	13,31	8,23	0,08	635
π_5	6	1348,24830	58,264	130 000	39 000	16,46	10,36	9,40	32 000

TAB. 11.2 – De gauche à droite : formule testée, nombre d'états de l'automate progressant, et pour chaque $i \in \{1, 2\}$, nombre d'états et de transitions de $\mathcal{M}^i_{\mathcal{A}}$ *avant* et *après* simplification, temps de calcul en secondes, mémoire utilisée en Ko.

11.5 L'outil PLTL2BA

Nous allons dans cette section parler plus en détail de l'outil PLTL2BA que nous avons conçu.

PLTL2BA est un programme développé dans le langage Ocaml. Son interface est très similaire à celle de LTL2BA : il prend en entrée une formule PLTL ainsi que des commutateurs optionnels permettant d'activer ou de désactiver certaines fonctionnalités, et affiche sur la sortie standard l'automate obtenu.

L'automate peut être affiché sous plusieurs formes : une description textuelle simple à lire, une représentation en Promela pour le logiciel SPIN, ou une troisème forme qui, envoyée vers programme DOT, produit automatiquement une image représentant l'automate.

L'outil n'étant pas encore entièrement finalisé (certaines fonctionnalités n'ont pas encore été implémentées et l'ergonomie doit être un peu améliorée), il n'est pas encore disponible au public. Lorsque la première version définitive du programme sera prête, nous la rendrons disponible, en proposant les sources sous licence GPL, et en créant une interface sur le modèle de celle que nous avons développée pour LTL2BA.

Chapitre 12

Conclusion et perspectives

Bilan de cette thèse

Dans cette thèse nous avons abordé le model-checking des logiques temporelles linéaires, en nous focalisant sur un point particulier : la transformation d'une formule de logique temporelle linéaire en un automate, le produit synchronisé de cet automate avec le modèle à vérifier permettant par un simple test du vide de déterminer si le modèle satisfait la spécification logique.

LTL

Pour pallier aux mauvaises performances techniques des model-checkers sur des formules LTL pourtant usuelles et de taille raisonnable, nous avons décidé d'explorer une voie jusqu'ici peu empruntée, car elle semblait peu performante : l'utilisation des automates alternants.

L'utilisation des automates alternants, de taille linéaire en la taille de la formule de départ, est clairement judicieuse. Le problème est que le coût de transformation d'un automate alternant en automate de Büchi est important. Nous avons alors trouvé une nouvelle méthode beaucoup plus efficace pour cette transformation, qui utilise le fait qu'un automate alternant issu d'une formule LTL est très faible.

Nous avons alors renforcé notre algorithme, en particulier en mettant l'accent sur les simplifications des automates, de façon à en réduire le nombre d'états et de transitions. L'utilisation d'automates de Büchi généralisés comme étape intermédiaire nous permet justement de pouvoir améliorer fortement l'efficacité de ces simplifications.

Nous avons alors implémenté cet algorithme, et cherché des moyens d'atteindre une grande efficacité pour nos calculs. C'est ce qui nous a amenés à utiliser la simplification à la volée pour réduire les ressources mémoire consommées, et à optimiser efficacement nos structures de données.

Le résultat est un outil, LTL2BA, dont les performances sont très intéressantes, et meilleures que celles de ses concurrents. De plus cet outil peut être utilisé en model-checking, puisqu'il est parfaitement compatible avec l'outil le plus répandu en model-

checking LTL : SPIN. Cet outil est disponible sous licence libre, et peut être utilisé par une interface web.

PLTL

Considérant l'apport considérable de notre algorithme dans le domaine du model-checking LTL, il nous a semblé parfaitement raisonnable de voir si ces améliorations pouvaient être adaptées à d'autres contextes. C'est alors que nous avons décidé d'aborder le cas de la logique temporelle avec opérateurs du passé, PLTL.

Les automates alternants à double sens sont aussi adaptés à PLTL que les automates alternants classiques le sont à LTL. Mais cette fois, le coût des transformations connues d'un automate alternant à double sens en un automate de Büchi est considérable, raison pour laquelle la voie du model-checking de PLTL reste peu explorée à cette date.

Nous avons alors utilisé le même raisonnement que pour LTL : chercher une sous-classe de ces automates pour laquelle la conversion vers les automates de Büchi est moins coûteuse. Cette classe, ce sont les automates progressants très faibles.

En raison des particularités impliquées par l'utilisation du passé, nous utilisons des automates de Büchi à mémoire comme étape intermédiaire de notre transformation, des automates à un seul sens qui se souviennent du passé proche.

L'algorithme que nous avons exposé pour PLTL a lui aussi fait l'objet d'une première implémentation, PLTL2BA. Toutefois nous ne lui connaissons aucun concurrent auquel il aurait pu être comparé.

Travaux futurs

Voici plusieurs pistes qui pourraient être explorées dans la continuation de cette thèse.

Tout d'abord nous devrions terminer l'outil PLTL2BA, en implémentant les quelques fonctionnalités manquantes, et le rendre disponible librement au public, comme LTL2BA, en diffusant les sources et en proposant une interface web. Ceci permettrait à d'autres personnes s'intéressant à ce domaine de comparer leur prototype à notre programme.

Ensuite, une voie de recherche nous paraît essentielle à explorer : effectuer directement le model-checking sur l'automate de Büchi généralisé (ou sur l'automate de Büchi à mémoire dans le cas de PLTL), sans générer inutilement un automate de Büchi. En effet générer un automate de Büchi pose un problème de choix dans l'ordre des conditions d'acceptation à satisfaire, et faire le test du vide directement sur le produit de l'automate de Büchi généralisé et du modèle permettrait de trouver plus efficacement de meilleurs contre-exemples.

Index

A

automate
- alternant . . . **38**, 38–49, 64–65
  - exécution, langage . . . **39**
  - exemples . . . 42
  - faible . . . **38**
  - optimisé . . . 65
  - simplification . . . 67–73
  - très faible . . . **38**
- alternant à double sens . . . **87**, 87–99
  - exécution bouclante . . . **89**
  - exécution progressante . . . **89**
  - exécution sans boucle . . . **89**
  - exécution, langage . . . **89**, **101**
  - exemple . . . 95
  - faible . . . **88**
  - optimisation . . . 121
  - progressant . . . **89**, 89–110, 112–118
  - sans boucle . . . **89**
  - simplification . . . 122–124
  - très faible . . . **88**
- de Büchi . . . **24**, 57–61, 119–120
  - exécution, langage . . . **24**
  - exemple . . . 24
  - optimisé . . . 67
  - simplification . . . 67–73, 122–124
- de Büchi à mémoire . . . **112**, 111–120, 124–126
  - exécution, langage . . . **112**
  - simplification . . . 122–124
- de Büchi généralisé . . . 29, **52**, 51–61, 112, 120
  - exécution, langage . . . **52**
  - exemples . . . 52
  - optimisé . . . 66
  - simplification . . . 67–73
- produit synchronisé . . . 26, 57
- représentation en mémoire . . . 63–67
- simplification . . . 67–73, 122–124

C

combinaisons booléennes positives . . . **37**, 64
complexité 24, 49, 56, 61, 83, 99, 110, 114, 118, 120

D

DOT . . . 78, 127

K

Kripke, structure de . . . **23**, 24–26
- exemple . . . 23

L

LTL . . . 13, 18–20, 44–49, 129–130
- exemples . . . 22
- forme normale négative . . . **19**
- sémantique . . . **18**
- simplification . . . 19–20
- syntaxe . . . **18**
- taille temporelle . . . **18**

LTL2BA . . . 34, 35, 73–78, 129

M

model-checking 10–13, 23–29, 61, 119–120, 129–130

P

PLTL . . . 20–22, 99–110, 130
- exemples . . . 22
- forme normale négative . . . **22**
- sémantique . . . **21**
- syntaxe . . . **21**
- taille temporelle . . . **21**

PLTL2BA . . . 85, 126–127, 130

Q

Q-DAG . . . **40**, 40–42
Q-forêt . . . **39**, **88**

S

SPIN . . . 29, 33, 34, 73–78, 127, 129

T

tableaux, méthode des . . . **27–29**, 84
- déclaratifs . . . 27–29, 83
- incrémentaux . . . 29, 33, 84

V

vide, test du . . . 25, **26**, 61, 120

Bibliographie

[Ari96] Ariane 5. Flight 501 failure. Report by the enquiry board, 1996.

[Bar95] G. Barrett. Model checking in practice : The T9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2) :69–78, February 1995. Special Section—Best Papers of FME (Formal Methods Europe) '93.

[BBF+00] S. Bensalem, M. Bozga, J-C. Fernandez, L. Ghirvu, and Y. Lakhnech. A transformational approach for generating non-linear invariants. In Jens Palsberg, editor, *Proceedings of the 7th International Symposium on Static Analysis (SAS 2000), Santa Barbara, CA*, volume 1824 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proceedings of the 4th International Symposium on Static Analysis (SAS 2000)*, volume 1302 of *Lecture Notes in Computer Science*, 1997.

[BH97] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of FIFO channel systems with nonregular sets of configurations (extended abstract). In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 560–570, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[Bir93] J-C. Birget. State-complexity of finite-state devices, compressibility and incompressibility. *Mathematical Systems Theory*, 26 :237–269, 1993.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293–318, September 1992.

[BVW94] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of the 6th International Computer Aided Verification Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, California, June 1994. Springer-Verlag.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of the 6th International Computer Aided Verification Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, 1994.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress of Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Berlin, June 1990. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM SIGACT and SIGPLAN, ACM Press, 1977.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons from branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, May 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications : A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGH94] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *Proceedings of the 6th International Computer Aided Verification Conference*, Lecture Notes in Computer Science, pages 415–427. Springer-Verlag, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, January 1978.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76 :96–120, 1988.

[CJM00] E.M. Clarke, S. Jha, and W. Marrero. Partial order reductions for security protocol verification. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1) :114–133, January 1981.

[CVWY91] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proceedings of Computer-Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Berlin, Germany, June 1991. Springer Verlag.

[DGV99] M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11th International Computer Aided Verification Conference*, pages 249–260, 1999.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I : Finite automata. *Journal of the ACM*, 41(3) :517–539, May 1994.

[DT98] C. Daws and S. Tripakis. Model-checking of real-time reachability properties using abstractions. In *TACAS'98, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329, Lisbon, Portugal, March 1998. Springer-Verlag.

[EH82] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In ACM, editor, *Proceedings of the fourteenth annual ACM Symposium on Theory of Computing, San Francisco, California, May 5–7, 1982*, pages 169–180, New York, NY, USA, 1982. ACM Press.

[EH00] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of 11th Int. Conf. on Concurrency Theory (CONCUR)*, 2000.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science, 1990.

[FHS95] A.R. Flora-Holmquist and M.G. Staskauskas. Formal validation of virtual finite state machines. In *Proceedings of the Workshop on Industrial-strength Formal Specification Techniques*. IEEE Computer Society Press, 1995.

[FJJM92] J. C. Fernandez, C. Jard, T. Jéron, and L. Mounier. On the fly verification of finite transition systems. *Formal Methods in System Design*, 1, 1992.

[FTM85] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *Proceedings of the Seventh International Symposium on Computer Hardware Description Languages and Their Applications*, pages 129–138, Amsterdam, 1985. IFIP.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Proceedings of the Fifth International Computer Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 1993.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001.

[GO03a] P. Gastin and D. Oddoux. LTL with past and two-way very-weak alternating automata. In *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'03)*, number 2747 in Lecture Notes in Computer Science, pages 439–448. Springer Verlag, 2003.

[GO03b] P. Gastin and D. Oddoux. LTL with past and two-way very-weak alternating automata. Technical Report LIAFA, Université Paris 7 (France), 2003. http ://www.liafa.jussieu.fr/∼gastin/Articles/gasodd03.html.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185, Berlin-Heidelberg-New York, 1990. Springer-Verlag.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (PoPL'80)*, pages 163–173, Las Vegas, Nev., 1980.

[GPVW95] R. Gerth, D.A. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2) :305–326, 1 May 1994.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant : A tutorial : Version 6.1. Technical Report RT-0204, Inria, Institut National de Recherche en Informatique et en Automatique, 1997.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, October 1969.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Hol97] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, May 1997.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996. Proceedings of the Second Spin Workshop.

[JJ89] C. Jard and T. Jéron. On-line model checking for finite linear temporal logic specifications. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of

Lecture Notes in Computer Science, pages 189–196, Grenoble, France, June 1989. Springer-Verlag.

[JLM+98] J. Julliand, B. Legeard, T. Machicoane, B. Parreaux, and B. Tatibouet. Specification of an integrated circuits card protocol application using B and Linear Temporal Logic. In *Proceedings of the 2nd Conference on the B Method*, April 1998.

[JLS96] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In *Proceedings of the Second SPIN Workshop*, Rutgers, Piscataway, NJ, USA, August 1996. American Mathematical Society.

[JMC94] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In T. Kropf and R. Kumar, editors, *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[Kam68] J.A.W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, California, 1968.

[KM97] J-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 239–258. Springer-Verlag, 1997.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proceedings of CAV'93*, number 697 in Lecture Notes in Computer Science, pages 97–109. Springer Verlag, 1993.

[KPV01] O. Kupferman, N. Piterman, and M.Y. Vardi. Extended temporal logic revisited. In *Proceedings of CONCUR'01*, number 2154 in Lecture Notes in Computer Science, pages 519–535. Springer Verlag, 2001.

[KV97] O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symposium on Theory of Computing and Systems ISTCS'97*, pages 147–158. IEEE, 1997.

[KVR83] R. Koymans, J. Vytopil, and W.P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 187–197, Montreal, Quebec, Canada, 17–19 August 1983.

[LMS02] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Logic in Computer Science*, pages 383–392, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[Lon93] D.E. Long. *Model-checking, abstraction and compositional verification.* PhD thesis, Carnegie Mellon University, 1993.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Verlag, Berlin, 1985.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148 :303–324, 1995.

[LS97] S. Loeffler and A. Serhouchni. Creating a validated implementation of the Steam Boiler control. *Proceedings of the Third SPIN Workshop*, April 1997.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurent systems : Specification.* Springer Verlag, Berlin-Heidelberg-New York, 1992.

[MS84] D. Muller and P. Schupp. Alternating automata on infinite objects : Determinacy and Rabin's theorem. In *Proceedings of the Ecole de Printemps d'Informatique Théoretique on Automata on Infinite Words*, volume 192 of *Lecture Notes in Computer Science*, pages 100–107, Le Mont Dore, France, May 1984. Springer Verlag.

[MS87] D. Muller and P. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3) :267–276, October 1987.

[MS95] D. Muller and P. Schupp. Simulating alternating tree automata by nondeterministic automata : New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1–2) :69–107, April 1995.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata. the weak monadic theory of the tree, and its complexity. In Laurent Kott, editor, *Automata, Languages and Programming, 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 275–283, Rennes, France, 15–19 July 1986. Springer-Verlag.

[MW81] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. Technical Report STAN–CS–81–872, Department of Computer Science, Stanford University, Stanford, CA 94305, September 1981.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1) :68–93, 1984.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS : A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on*

Automated Deduction (CADE-11), volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer-Verlag.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the Sixth International Computer Aided Verification Conference (CAV'94)*, Lecture Notes in Computer Science, pages 377–390. Springer-Verlag, 1994.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. In *Theoretical Computer Science*, volume 13, pages 45–60, 1981.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI series in Computer and System Sciences*, pages 123–144, New York, 1984. Springer-Verlag.

[Pnu86] A. Pnueli. Specification and development of reactive systems. In *Information processing 86 : proceedings of the 10th IFIP World Computer Congress*, IFIP congress series, Dublin, Ireland, September 1986. North-Holland.

[Pri51] A.N. Prior. *Past, Present and Future*. Oxford University Press, Oxford, 1951.

[PS81] A. Pnueli and R. Sherman. Semantic tableau for temporal logic. Technical Report CS81–21, The Weizmann Institute, 1981.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, Berlin/New York, 1982.

[RL97] T. Ruys and R. Langerak. Validation of Bosch' mobile communication network architecture with Spin. *Proceedings of the Third SPIN Workshop*, April 1997.

[RMD+92] Y.S. Ramakrishna, L.E. Moser, L.K. Dillon, P.M. Melliar-Smith, and G. Kutty. An automata theoretic decision procedure for propositional temporal logic with Since and Until. *Fundamenta Informaticae*, 17 :271–282, 1992.

[Roh97] S. Rohde. *Alternating Automata and the Temporal Logic of Ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, Berlin, 1971.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV : International Conference on Computer Aided Verification*, 2000.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3) :733–749, 1985.

[Sch97] K. Schneider. CTL and equivalent sublanguages of CTL*. In C. Delgado Kloos, editor, *Proceedings of International Conference on Computer Hardware Description Languages and their Applications*, pages 40–59, Toledo,Spain, April 1997. IFIP, Chapman and Hall.

[Sch01] K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. *Lecture Notes in Computer Science*, 2250 :39–55, 2001.

[Sis83] A.P. Sistla. *Theoretical Issues in the Design of Distributed and Concurrent Systems*. PhD thesis, Harvard University, Cambridge, MA, 1983.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, June 1972.

[Tau99] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Workshop Concurrency, Specifications and Programming*, pages 251–262, Warsaw, Poland, 1999.

[Var88] M. Y. Vardi. A temporal fixpoint calculus. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 250–259, New York, NY, USA, 1988. ACM Press.

[Var96] M.Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Verlag, New York, NY, USA, 1996.

[Var98] M.Y. Vardi. Reasonning about the past with two-way automata. In *Proceedings of ICALP'98*, number 1443 in Lecture Notes in Computer Science, pages 628–641. Springer Verlag, 1998.

[Var01] Moshe Y. Vardi. Branching vs. linear time : Final showdown. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, April 2001.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[VW94] M.Y. Vardi and P. Wolper. Reasonning about infinite computations. *Information and Computation*, 115 :1–37, 1994.

[Win90] G. Winskel. Compositional checking of validity on finite state processes. *Workshop on theories of communication*, 458, 1990.

[Wol01] P. Wolper. Constructing automata from temporal logic formulas : A tutorial. *Lecture Notes in Computer Science*, 2090 :261–278, 2001.

[WVS83] P. Wolper, M.Y. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.