Thèse de doctorat de l'Université Pierre et Marie CURIE
Paris 6

Spécialité
Informatique

Présentée par
Alexandre DURET-LUTZ

Pour obtenir le grade de
Docteur de l'Université Pierre et Marie CURIE

Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents

Soutenue le 10 juillet 2007 devant un jury composé de :

M. **Ahmed BOUAJJANI**, Professeur à l'Université Paris 7	Rapporteur
M. **François VERNADAT**, Professeur à l'INSA de Toulouse	Rapporteur
M. **Jean-Michel COUVREUR**, Professeur à l'Université d'Orléans	Examinateur
M. **Claude GIRAULT**, Professeur émérite à l'Université Paris 6	Examinateur
M. **Alain GRIFFAULT**, Maître de conférences à l'Université Bordeaux 1	Examinateur
M. **Serge HADDAD**, Professeur à l'Université Paris-Dauphine	Examinateur
M. **Fabrice KORDON**, Professeur à l'Université Paris 6	Directeur de thèse
M. **Denis POITRENAUD**, Maître de conférences à l'Université Paris 5	Encadrant

Laboratoire d'Informatique de l'Université Paris 6

Corrections. Ce document est très légèrement différent de la version que j'ai remise aux rapporteurs et à l'université avant ma soutenance, car je continue d'y corriger les menues erreurs que j'y découvre régulièrement lorsque je l'ouvre.

Ces corrections, principalement orthographiques, sont bien trop mineures pour être listées ici. Le lecteur intéressé pourra consulter l'historique du dépôt `git` mentionné ci-dessous. Le numéro de version de ce document est indiqué au bas de cette page.

N'hésitez par à me signaler toute erreur à `<adl@gnu.org>`.

Sources LaTeX. Si l'on met de côté les planches d'animation des algorithmes (pages 111, 122, 127, 170, 171 et 114) et la bibliographie, l'ensemble de ce document est compilé à partir d'un unique fichier LaTeX contenant texte et figures.

Si vous souhaitez réutiliser des parties du texte ou certaines figures, les sources de ce document peuvent-être obtenues à partir d'un dépôt `git` de la façon suivante :

```
% git clone http://www.lrde.epita.fr/~adl/git/th.git
% cd th
```

(Le gestionnaire de versions `git` peut être obtenu depuis `http://git.or.cz/`.)

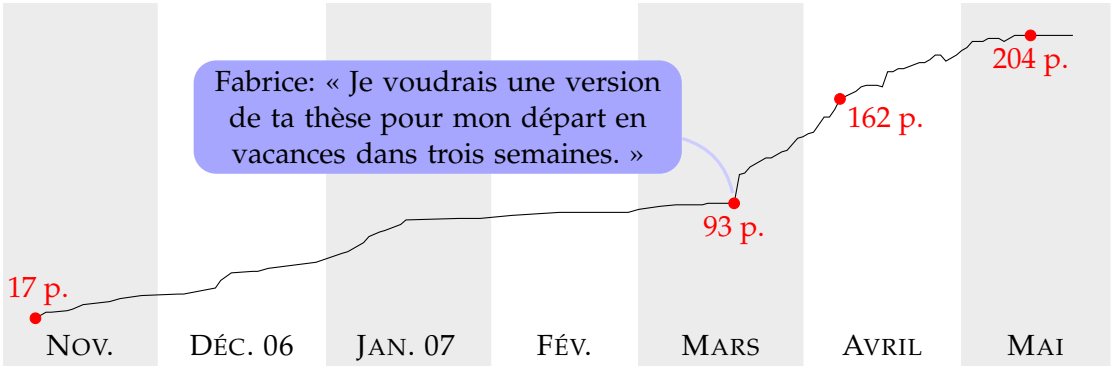
Le tag `version-imprimee` indique la version qui a été imprimée par l'université et distribuée lors de ma soutenance. Le tag `soutenance` donne la version qui était en ligne sur ma page web lors de la soutenance.

Remerciements

Je ne conçois de commencer cette page qu'en remerciant Thierry GÉRAUD et Akim DEMAILLE. Je n'en serais pas là s'ils ne m'avaient détourné de mon bonhomme de chemin il y a 9 ans pour me faire plonger dans le monde de la recherche et dans celui des logiciels libres.

De façon beaucoup plus actuelle, je tiens à remercier messieurs Ahmed BOUAJJANI et François VERNADAT, qui ont bien voulu rapporter cette thèse malgré une fin d'année très chargée, ainsi que Jean-Michel COUVREUR, Claude GIRAULT, Alain GRIFFAULT et Serge HADDAD qui ont accepté de faire partie de mon jury. Jean-Michel est un puits d'idées auquel cette thèse doit énormément ; l'article qui y est le plus cité est naturellement l'un des siens [34], avec 23 occurrences[1].

Cette thèse, dirigée par Fabrice KORDON et encadrée par Denis POITRENAUD est la prolongation de mon stage de DEA encadré par Denis POITRENAUD et Jean-Michel ILIÉ. Je leur suis reconnaissant à tous les trois. Travailler avec Denis est un plaisir ; j'admire sa patience et sa façon d'aller droit au but. Quant à Fabrice, son influence sur la rédaction de cette thèse ne saurait être mieux illustrée que par la courbe (authentique) suivante:



Après Denis, la personne avec laquelle j'ai le plus travaillé est Soheib BAARIR. Merci pour les heures que nous avons passées ensemble (à travailler ou non) et que le chapitre que nos thèses partagent est loin de résumer.

Pour en finir avec les remerciements relatifs à ce document, je veux exprimer ma gratitude à tout ceux qui ont participé à sa relecture. Denis POITRENAUD a eu la chance de découvrir tous les chapitres dans le désordre, j'ignore où il puisait le courage de relire minutieusement des chapitres dont il connaissait pourtant déjà la teneur. Merci aussi à mes parents, qui ont chacun relu une version quasi intégrale en étant fiers de ne rien comprendre et visiblement pas trop effrayés par les monstruosités syntaxiques qu'ils ont relevées. Merci enfin à tous ceux qui ont relu un ou plusieurs chapitres: Cédric BESSE, Akim

[1] Je compte celle-ci, ce qui me permet de faire en sorte qu'elle constitue la première citation du document.

DEMAILLE, Fabrice KORDON, Emmanuel PAVIOT-ADET et Sébastien PÉREZ-DUARTE. J'ai naturellement oublié certaines de leurs corrections et changé de nombreux passages après leurs relectures afin qu'ils ne puissent pas s'attribuer la moindre de mes erreurs (ce serait trop facile).

Je veux aussi remercier Heikki TAURIAINEN pour les mails que nous avons échangés, ainsi que Keijo HELJANKO pour l'accueil qu'ils m'ont tous les deux réservé à l'Université Technologique d'Helsinki en septembre 2006.

Outre ce que j'ai appris d'un point de vue scientifique durant cette thèse, l'un des points les plus positifs de ces quatre dernières années a été mon expérience d'enseignement à l'université Paris 6. Merci à ceux avec qui j'ai eu le plaisir d'enseigner: Claude DUTHEILLET, Fabrice LEGOND-AUBRY, Valérie MENISSIER-MORAIN, Isabelle MOUNIER, Guénaël RENAULT et Philippe TREBUCHET.

Cette thèse s'est déroulée au LIP6 où certains ingénieurs systèmes ou personnels administratif sont aussi compétents que gentils et disponibles. Je souhaite remercier en particulier Nicolas GIBELIN, Raymonde KURINCKX, Thierry LANFROY et Jean-Luc MOUNIER.

Merci aux thésards qui ont occupé les bureaux C1148 ou 818 à un moment ou à un autre : Soheib BAARIR, Alexandre HAMEZ, Alban LINARD, Xavier RENAULT, Yann THIERRY-MIEG et Jean-Baptiste VORON (leur présence était loin d'être aussi désagréable que j'ai pu le laisser supposer lorsque je rédigeais...) ainsi qu'à l'ensemble des (trop nombreux) joueurs de blitz à quatre de l'étage: Soheib BAARIR, Réda BENDRAOU, Xavier BLANC, Cédric BESSE, Frédéric GILLIERS, Maher LAMARI, Xavier RENAULT, Julien « j'ai une question » SOPENA, Pierre SUTRA, Yann THIERRY-MIEG, Tewfik ZIADI et Mikal ZIANE (vous voilà dénoncés).

Pour terminer, je souhaite remercier la communauté des *bookcrossers* francophones, toujours de bonne humeur et jamais à court d'encouragements, ainsi que les panoramistes parisiens que j'ai rencontrés cette année, habitués du regard circulaire. Merci en particulier à Sébastien PÉREZ-DUARTE dont le mail d'encouragements ne pouvait pas mieux tomber.

Table des matières

Remerciements . . . 3
Principales notations . . . 9

1 Introduction générale **11**
1.1 Enjeux de la vérification . . . 11
1.2 Méthodes formelles . . . 12
1.3 Contexte de travail . . . 13
1.4 Objectifs de la thèse . . . 14
1.5 Plan du mémoire . . . 15

2 Positionnement et exemple **17**
2.1 Système et modèle . . . 17
2.2 Espace d'état . . . 19
2.3 Formule LTL . . . 19
2.4 Traduction de la formule en automate . . . 22
2.5 Produit et *emptiness check* . . . 23
2.6 Hypothèses d'équité . . . 27
2.7 Explosion combinatoire . . . 27
2.8 Résumé de l'approche automate . . . 28

3 Automates de Büchi basés sur les transitions **31**
3.1 Séquences et propositions atomiques . . . 31
3.1.1 Propositions atomiques . . . 31
3.1.2 Séquences . . . 32
3.1.3 Expressions ω-rationnelles . . . 34
3.1.4 Séquences infinies sur 2^{AP} . . . 34
3.1.5 Formalismes de représentation de langages . . . 35
3.2 Logiques temporelles . . . 36
3.2.1 F1S: Logique monadique du premier ordre à un successeur . . . 37
3.2.2 LTL: Logique temporelle à temps linéaire . . . 37
3.2.3 S1S: Logique monadique du second ordre à un successeur . . . 41
3.3 ω-Automates . . . 42
3.3.1 Structure de Kripke . . . 42
3.3.2 Automates de Büchi . . . 43
3.3.3 Automates de Büchi généralisés . . . 45
3.3.4 Conditions d'acceptation sur les transitions: TGBA . . . 48
3.3.5 Lien entre TGBA et GBA . . . 51
3.3.6 Opérations sur les TGBA . . . 53
3.3.7 Vérification des séquences finies . . . 57

3.4 Conclusion 62

4 Traduction de formules LTL **65**
4.1 Existant 65
4.1.1 Constructions combinatoires 65
4.1.2 Constructions par composition 68
4.1.3 Constructions par tableau 74
4.1.4 Constructions par automates alternants 80
4.2 Traduction de J.-M. Couvreur 86
4.3 Améliorations 91
4.3.1 Amélioration du déterminisme 91
4.3.2 Branching postponement 93
4.3.3 Simplifications de formules 94
4.3.4 Étude des composantes fortement connexes 97
4.4 Comparatif 98
4.5 Une idée: prendre en compte la logique des promesses 102
4.6 Conclusion 103

5 Emptiness checks de TGBA **105**
5.1 Introduction 105
5.2 Principe et historique 105
5.3 Parcours en profondeur imbriqués 108
5.3.1 Le cas non généralisé 109
5.3.2 Ajout des poids 112
5.3.3 Cas généralisé 115
5.4 Utilisation des composantes fortement connexes 118
5.4.1 Algorithme de Couvreur 120
5.4.2 Heuristiques 127
5.5 Étude de performances 128
5.6 Calcul de contre-exemples 136
5.6.1 Contre-exemples pour Cou99 137
5.6.2 Contre-exemples pour Tau03 139
5.6.3 Mesures des contre-exemples 139
5.7 *Bit-state hashing* 140
5.8 Conclusion 142

6 Emptiness check avec inclusion **143**
6.1 Introduction 143
6.2 Définitions 145
6.3 Emptiness check d'un $\wp$-TGBA 148
6.3.1 Adaptation aux $\wp$-TGBA 148
6.3.2 Nouvel algorithme 149
6.3.3 Contre-exemples 151
6.4 *Emptiness check* semi-décisionnel 152
6.5 Application aux symétries 153
6.5.1 Construction d'un $\wp$-TGBA basée sur les symétries 154
6.5.2 Structures et opérations pour l'*emptiness check* 155
6.5.3 Évaluation 155

6.6 Conclusion 158

7 Hypothèses d'équité **159**
7.1 Introduction 159
7.2 Définitions 160
7.3 Automates de Streett 164
7.4 Cas pratique 166
7.5 Équité pour les systèmes de transitions synchronisés 167
7.6 Emptiness Check d'automates de Streett 168
7.7 Conclusion 179

8 Conclusion générale **181**
8.1 De l'intérêt des TGBA 181
8.2 Optimisations et nouveaux algorithmes 182
8.3 Perspective à court terme: Ensembles persistants équitables 182
8.4 Vers une parallélisation de l'*emptiness check* 185
8.5 Problème ouvert: traduction efficace de LTL vers Streett 186

Bibliographie **187**

A Spot **201**
A.1 Bibliothèque centrée sur les TGBA 201
A.2 Support des calculs à la volée 201
A.2.1 `tgba` utilisé comme interface pour des extensions 203
A.3 `tgba` comme un formalisme d'entrée 204
A.4 Disponibilité et utilisations connues 205

Résumé **206**

Principales notations

AP	ens. des propositions atomiques	sec. 3.1.1 page 31
2^{AP}	ens. des valuations de AP	sec. 3.1.1 page 31
$2^{2^{AP}}$	ens. des formules propositionnelles	sec. 3.1.1 page 31
$\mathbb{N}$, ω	entiers naturels, premier ordinal infini	déf. 1 page 32
$[\![n, m]\!]$, $]\!]n, m[\![$	intervalles d'entiers	déf. 2 page 32
σ, $\sigma(i)$, σ^j	séquence, $(i+1)^{\text{e}}$ élément, suffixe	sec. 3.1.2 page 32
$a \cdot b$, $a + b$, a^ω, a^*	expressions ω-rationnelles	sec. 3.1.3 page 34
$\sigma_{\|AP'}$, $L_{\|AP'}$	projection d'une séquence, d'un ens.	déf. 11 page 35
$\sigma \parallel \sigma'$	composition de séquences	déf. 12 page 35
$\mathsf{F}\,\varphi$, $\mathsf{G}\,\varphi$, $\varphi\,\mathsf{R}\,\psi$, $\varphi\,\mathsf{U}\,\psi$, $\mathsf{X}\,\varphi$	opérateurs LTL	sec. 3.2.2 page 37
$\mathscr{L}_{AP}(\varphi)$	langage d'une formule LTL	sec. 3.2.2 page 37
$A = \langle AP, Q, Q^0, \mathcal{F}, \delta \rangle$	TGBA	déf. 27 page 48
$t = (t^{\text{in}}, t^{\text{prop}}, t^{\text{acc}}, t^{\text{out}}) \in \delta$	transition d'un TGBA	sec. 3.3.4 page 48
$\text{Run}(A)$, $\text{Acc}(A)$, $\mathscr{L}(A)$	chemins, chemins acceptants, langage	déf. 28 page 49
$\text{Reach}(A)$	états accessibles de $\mathcal{A}$	déf. 29 page 49
$A[\mathcal{T}]$	substitution d'états initiaux	déf. 30 page 50
$A \otimes A'$	produit synchronisé d'automates	déf. 31 page 53
A_φ	automate tel que $\mathscr{L}(A_\varphi) = \mathscr{L}_{AP}(\varphi)$	chap. 4 page 65

Chapitre 1

Introduction générale

Ce premier chapitre situe ce travail au sein du large domaine qu'est la vérification. L'objectif est d'introduire l'*approche automate du model checking* comme une technique particulière de vérification. Cette approche, sur laquelle repose l'ensemble de cette thèse, sera développée et illustrée dans le chapitre 2.

1.1 Enjeux de la vérification

De la machine à laver aux systèmes de contrôle aérien, en passant par les feux tricolores, les centraux téléphoniques et les terminaux de paiement, nous sommes entourés de systèmes plus ou moins automatiques sur lesquels nous comptons. Certains de ces systèmes sont plus critiques que d'autres et leurs erreurs peuvent avoir des conséquences importantes (dégradation des performances, pertes de données, pertes financières, dommages physiques) que l'on souhaite éviter dès la conception.

Soumettre un système à une batterie de tests, afin de s'assurer qu'il se comporte correctement dans un certain nombre de scénarii, est certes une expérience rassurante, mais elle ne garantit rien sur le comportement du système dans tous les autres cas. La génération automatique de tests de façon à maximiser la couverture du système est, du reste, un domaine de recherche qui bénéficie des méthodes formelles que nous introduirons dans la section suivante [40, 131].

Ici, nous visons une vérification *exhaustive* de la totalité des comportements possibles en prouvant que chacun est correct. L'exhaustivité ne peut être atteinte qu'en travaillant à partir d'un modèle et cela a l'avantage de pointer des erreurs dès la conception du modèle.

Nous nous intéressons plus particulièrement à des systèmes qui peuvent être vus comme un ensemble de composants travaillant parallèlement, tout en interagissant occasionnellement. Les applications client/serveur en sont des exemples typiques. Le parallélisme introduit par ces systèmes et leur complexité augmentent le nombre de scénarii possibles et contribuent à rendre la vérification difficile.

1.2 Méthodes formelles

Les méthodes formelles [31] sont des langages, techniques et outils mathématiques permettant de spécifier et vérifier des systèmes. On construit d'un côté un *modèle* du système dans un formalisme donné, et l'on exprime par ailleurs un ensemble de propriétés que l'on souhaite prouver sur ce système.

De façon grossière, on peut dire qu'il existe deux types de méthodes formelles pour prouver qu'un modèle vérifie un ensemble de propriétés: la preuve de théorèmes ou l'exploration de l'ensemble des états du système.

Preuve de théorèmes. Pour faire de la preuve de théorèmes, le modèle et les propriétés à prouver sont exprimés sous formes de formules logiques. Ces formules peuvent être combinées entre elles en respectant des règles définies (un système formel) pour déduire de nouvelles formules. Les preuves de théorèmes se font donc par dérivation à partir des propriétés connues du modèle.

Tout le problème est d'automatiser ces constructions de preuves. En pratique, les outils de preuve de théorèmes ne sont pas entièrement automatiques: ils requièrent l'aide interactive de l'utilisateur et leur emploi demande une bonne expertise. D'autre part, si ces preuves permettent d'établir la correction d'un modèle, elles se révèlent peu pratiques en cas d'erreur: si une propriété n'est pas vérifiée, il peut être difficile de trouver le comportement du système qui en est la cause.

Un avantage non négligeable de cette approche est qu'elle peut s'appliquer à des modèles paramétrés (prouver une propriété « *pour tout n* », n représentant par exemple le nombre de clients dans un modèle d'application client/serveur).

Le système Météor [14] qui équipe la ligne 14 du métro parisien est un exemple phare de système complexe dont les composants critiques ont été développés par la méthode B [1] en s'appuyant sur des preuves logiques.

Exploration de l'ensemble des états. Les méthodes basées sur l'exploration de l'ensemble des états, désignées aussi sous le nom de *model checking* (vérification de modèle[1]) procèdent différemment [94, 30, 74, 80]. Ces méthodes reposent sur une modélisation du système appelée *espace d'état*, décrivant tous les états accessibles du système ainsi que les liens entre eux. Une telle structure est généralement immense, voire infinie, mais a l'avantage de pouvoir être construite automatiquement à partir d'un modèle de plus haut niveau. Par exemple l'espace d'état d'un système modélisé sous la forme d'un réseau de Petri correspond à son graphe des marquages accessibles [41].

De nombreuses propriétés peuvent alors être vérifiées algorithmiquement sur cet espace d'état. Le travail de l'utilisateur se réduit à modéliser le système, spécifier les propriétés à vérifier, puis à lancer le calcul. Ces méthodes, parce qu'elles explorent l'ensemble des états du système, sont en général capables d'exhiber des contre-exemples lorsque la pro-

[1] Nous avons choisi d'employer les expressions anglaises lorsque leurs équivalents français ont un sens moins précis (« *model checking* » a un sens précis trop largement recouvert par « vérification de modèle ») ou sont trop peu usités (« *emptiness check* » au lieu de « test de vacuité »).

priété recherchée n'est pas vérifiée. Ces caractéristiques, simplicité d'utilisation et retour sur erreur, sont les points forts du *model checking* vis-à-vis de la preuve de théorèmes.

En revanche, et à la différence de la preuve de théorèmes, le *model checking* ne considère que des systèmes à ensemble d'états fini, c'est-à-dire des systèmes ne pouvant atteindre qu'un nombre fini de configurations. Cette restriction est due au fait que ces méthodes procèdent par exploration de l'ensemble des états accessibles. Par exemple les circuits logiques et certains protocoles de communication sont des systèmes à états finis. Un programme contenant une variable non bornée n'est évidement pas fini.

Nous ne nous intéresserons pas aux techniques qui visent à ramener un système infini à une abstraction finie. Esparza [47] donne une courte bibliographie sur la vérification de systèmes dont l'ensemble d'états est infini. Toutefois les travaux que nous présentons peuvent aussi s'appliquer à la vérification partielle d'un système infini (en le bornant).

C'est à cette dernière classe de méthodes formelle que nous allons nous intéresser ici. D'une certaine façon, on peut considérer que le *model checking* désigne la branche entièrement automatique des méthodes formelles.

Dans le cas particulier où le modèle est formalisé sous la forme d'un réseau de Petri, nous pouvons considérer une troisième approche: celle des *méthodes structurelles*. Il s'agit d'algorithmes qui peuvent établir certaines propriétés, comme le fait que les places du réseau soient bornées, en étudiant la structure du réseau et sans jamais développer son espace d'état. D'autres algorithmes existent pour établir des propriétés sur des sous-structures locales du réseau, comme les P et T-invariants, les trappes ou les verrous. Si ces techniques ne visent pas à prouver les propriétés spécifiées par l'utilisateur, elles peuvent servir de préliminaires en preuve de théorèmes comme en *model checking*. Par exemple le *model checking* suppose un espace d'état fini alors que les réseaux de Petri peuvent représenter des espaces d'état infinis: il est possible de vérifier structurellement si un réseau de Petri est borné, ce qui implique que son espace d'état est fini.

1.3 Contexte de travail

Pour préciser davantage notre position, caractérisons maintenant les systèmes et le type de propriétés que nous souhaitons y vérifier.

Systèmes réactifs, finis et fermés. Nous considérons les systèmes dits « *réactifs* », qui interagissent avec leur environnement de façon continue. Une application dans laquelle des clients et des serveurs communiquent par messages de façon continue est un système réactif. Un programme qui calcule la n^{e} décimale de π avant de se terminer n'est pas un système réactif.

Les systèmes fermés sont ceux dans lesquels l'environnement est représenté comme un composant du système. Le modèle est complet au sens où toutes les interfaces possibles sont modélisées.

Comme on l'a dit, les systèmes vérifiés en *model checking* doivent être finis. La plupart du temps nous ne nous occuperons pas du formalisme de haut niveau utilisé pour modéliser le système et ne travaillerons qu'à partir de son espace d'état.

Les modèles que nous considérons ont beau avoir un nombre d'états fini, leurs exécutions (séquences d'états) ne se terminent pas et sont donc infinies, de plus elles peuvent être en nombre infini. Nous tenons à vérifier des propriétés sur toutes les exécutions possibles d'un système, de façon exhaustive, mais on conçoit que cette vérification ne se fera pas en considérant les exécutions possibles une à une de façon explicite!

Logique temporelle à temps linéaire. CTL (*Computational Tree Logic*) et LTL (*Linear Temporal Logic*) sont deux logiques temporelles propositionnelles parmi les formalismes couramment utilisés pour exprimer des propriétés. En plus des opérateurs logiques classiques ($\wedge$, $\neg$, ...), ces logiques introduisent des opérateurs temporels pour spécifier des propriétés qui concernent les successeurs d'un état dans le temps.

CTL est une logique à temps arborescent dans laquelle chaque état est vu comme ayant plusieurs successeurs (c'est-à-dire plusieurs futurs) possibles. LTL, en revanche, est une logique à temps linéaire où chacune des séquences d'exécution possibles du système est considérée indépendamment: un état n'a donc qu'un successeur au sein d'une exécution.

Les expressivités de CTL et LTL ne sont pas comparables [137]. Il existe des propriétés exprimables dans les deux logiques, mais il existe aussi des propriétés qui ne sont exprimables que dans l'une ou l'autre. L'avantage de CTL est algorithmique : la vérification d'une formule CTL peut se faire en un temps linéaire par rapport à la taille de la formule, tandis que la vérification d'une formule LTL peut prendre un temps exponentiel par rapport à cette même taille. La logique LTL passe pour être plus intuitive pour l'utilisateur (parce que le temps n'y est pas arborescent) et permet d'exprimer directement certaines propriétés importantes comme l'*équité* (nous y reviendrons). D'autre part, contrairement à celui qui utilise LTL, le *model checking* utilisant CTL ne produit pas de contre-exemple sous la forme de séquence d'états (des exécutions qui ne vérifient pas la propriété) mais sous la forme d'un ensemble d'états (des situations où la propriété n'est plus vérifiée).

Par la suite nous nous intéresserons exclusivement à la logique LTL, pour laquelle le problème de la vérification peut se ramener à des opérations sur des automates.

1.4 Objectifs de la thèse

Approche automate. L'approche automate pour le *model checking* de propriétés LTL est une technique classique de vérification formelle de modèles de systèmes concurrents [136, 139]. Un modèle S, ainsi qu'une propriété φ qui doit y être vérifiée, sont modélisés sous forme d'automates de Büchi. Ces automates acceptent des mots de longueur infinie, ce qui nous permet de modéliser l'ensemble des exécutions d'un modèle par un automate A_M et l'ensemble des exécutions qui invalident la formule par un autre automate $A_{\neg\varphi}$. En calculant le *produit synchronisé* de ces deux automates, nous obtenons l'automate $A_M \otimes A_{\neg\varphi}$ représentant l'intersection: l'ensemble des exécutions du modèle qui invalident la formule. Cependant, à cause de la structure particulière des automates de

Büchi, déterminer si l'ensemble des mots acceptés par un automate est vide requiert un algorithme dédié. Dans cette approche le modèle M vérifie la formule φ si et seulement si l'automate produit $A_M \otimes A_{\neg\varphi}$ n'accepte aucun mot.

Utilisation d'automates de Büchi généralisés basés sur les transitions. Dans ce mémoire nous revisitons cette approche en utilisant un type d'automates de Büchi particulier: des automates de Büchi *généralisés* et *basés sur les transitions* (TGBA). Ces automates sont plus concis que ceux utilisés traditionnellement, et cela profite au processus de vérification. Dans un premier temps, nous étudierons les différents algorithmes existants pour effectuer les deux étapes principales de l'approche automate: la traduction d'une formule LTL en automate de Büchi et l'*emptiness check* (test de vacuité) d'un automate. Tout en apportant des améliorations aux différentes techniques existantes, nous montrerons qu'il est intéressant de privilégier l'utilisation des TGBA.

Variantes de l'*emptiness check* pour automates ensemblistes et hypothèses d'équité. Dans un deuxième temps, nous étudierons deux variantes de cette approche du *model checking*.

L'une concerne un cas particulier de vérification où le modèle a été réduit par regroupement de ses états. Nous proposerons alors d'adapter l'algorithme d'*emptiness check* pour tirer parti de ces regroupements d'états.

L'autre permet de prendre en compte des hypothèses d'équité. Il s'agit d'hypothèses qui restreignent l'interprétation du modèle en supprimant certaines exécutions « non équitables ». Nous distinguerons des hypothèses d'équité faible, qui peuvent être aisément représentées par un TGBA, et les hypothèses d'équité forte qui demandent l'emploi d'un autre type d'automates: les automates de Streett. Nous proposerons alors un nouvel algorithme d'*emptiness check* pour ces automates.

1.5 Plan du mémoire

Le prochain chapitre illustrera par un exemple concret l'approche automate pour le *model checking* de propriétés LTL que nous avons évoqué dans la section précédente. Ce sera l'occasion de montrer les étapes importantes de l'approche: la traduction d'une formule LTL en TGBA et l'*emptiness check* d'un TGBA, puis de souligner l'importance de réduire la taille de l'automate produit. Ce chapitre est volontairement informel.

Le chapitre 3 introduira les principaux concepts et objets que nous manipulons. Principalement: la logique temporelle à temps linéaire (LTL) pour représenter φ, les structures de Kripke pour représenter A_M, les automates de Büchi généralisés basés sur les transitions (TGBA) pour représenter $A_{\neg\varphi}$, et le produit synchronisé d'automates.

Le chapitre 4 se concentrera sur les techniques de traduction d'une formule LTL en TGBA. Après un large état de l'art, nous nous attarderons sur une traduction proposée par Couvreur [34] à laquelle nous apporterons plusieurs améliorations. Le chapitre sera conclu par une comparaison expérimentale des différents traducteurs disponibles.

Le chapitre 5, quant à lui, passera en revue les techniques permettant de déterminer si le langage d'un TGBA est vide ou non, et de calculer un contre-exemple le cas échéant. Ici encore nous suivrons un plan similaire à celui du chapitre précédent: nous proposerons des améliorations pour les deux classes d'algorithmes d'*emptiness check* existants, et nous comparerons les divers algorithmes de façon expérimentale. Cette comparaison vise à montrer qu'il est intéressant d'utiliser les TGBA d'un bout à l'autre du *model checking*, plutôt que de travailler avec des automates *dégénéralisés* comme cela est fait habituellement.

Les chapitres 6 et 7 introduiront tous les deux des variantes d'*emptiness check* pour des approches légèrement différentes du *model checking* par automates:

– Dans le chapitre 6, nous nous intéresserons au cas où la génération de l'espace d'état produit un automate dont les états représentent des ensembles d'états du modèle. Ce type d'automate apparaît dans certaines techniques de réductions qui cherchent à réduire la taille de l'espace d'état en regroupant les états. L'*emptiness check* est changé pour effectuer des tests d'inclusion et diriger la construction à la volée de l'espace d'état.
– Dans le chapitre 7, nous étudierons différentes façons de prendre en compte des hypothèses d'équité dans cette approche. Pour gérer des hypothèses d'équité forte, nous introduirons des automates de Streett, montrerons comment la génération de l'espace d'état peut être changée pour produire un tel automate, et proposerons un nouvel algorithme d'*emptiness check* pour ces automates. (La traduction de formules LTL reste inchangée car les automates de Büchi peuvent être facilement convertis en automates de Streett.)

Tous les algorithmes est techniques décrits dans ce mémoire ont été implémentés dans une bibliothèque, Spot, que nous décrirons succinctement dans l'annexe A.

Chapitre 2

Positionnement et exemple

Ce chapitre illustre de façon informelle les différentes étapes de l'approche automate du *model checking* sur un petit exemple. L'exemple est volontairement simpliste afin de pouvoir développer entièrement son espace d'état.

Les sections de ce chapitre correspondent à des étapes de l'approche, ou à des variantes qui seront discutées plus amplement par la suite.

Les dernières sections de ce chapitre résument les différentes étapes présentées tout en annonçant le plan.

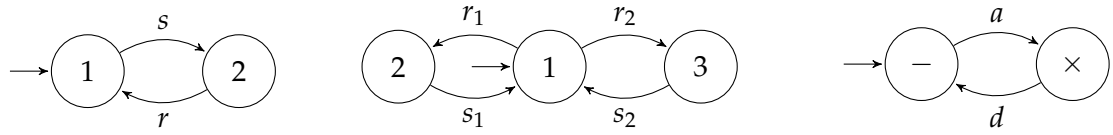
2.1 Système et modèle

Considérons un système simple constitué de deux clients C_1 et C_2 qui échangent des messages avec un serveur S. Un client peut envoyer une requête au serveur, il doit ensuite attendre la réponse du serveur avant de pouvoir envoyer une nouvelle requête. On suppose que les messages transitent sur des canaux sans perte, que les envois et réceptions se font de façon asynchrone (un message peut être envoyé si le récepteur n'est pas en train de l'attendre), et que l'ordre d'arrivée des messages n'est pas garanti: si les deux clients envoient une requête dans un certain ordre, le serveur ne les recevra pas forcément dans cet ordre.

On souhaite raisonner sur ces envois et réceptions de messages pour exprimer des propriétés telles que « *si un client envoie une requête, il recevra forcément une réponse* ».

Plusieurs formalismes de haut niveau peuvent être utilisés pour modéliser de tels systèmes. Par exemple des réseaux de Petri [41], un programme Promela [75], ou encore des automates communicants [21]. Ici nous avons choisi de modéliser ce système avec des systèmes de transitions synchronisés [6]. Ce choix a peu d'influence sur l'approche suivie: l'important est de pouvoir ensuite utiliser ce modèle pour générer un espace d'état fini. Générer l'espace d'état est possible avec tous les formalismes cités, mais pour certains il faudra d'abord prouver que cet espace est fini.

La figure 2.1 page suivante montre des automates modélisant les différents sous-systèmes de notre exemple. Un client possède deux états: il passe de l'un à l'autre en envoyant (*s*



(a) C: un client (b) S: le serveur (c) B: un canal

FIG. 2.1: Modèle des sous-systèmes qui seront synchronisés pour construire un modèle du système complet.

(1)	$\langle s, ., ., ., ., ., ., a, . \rangle$
(2)	$\langle ., s, ., ., ., ., ., a \rangle$
(3)	$\langle r, ., ., ., d, ., ., ., . \rangle$
(4)	$\langle ., r, ., ., ., d, ., ., . \rangle$
(5)	$\langle ., ., ., r_1, ., ., ., d, . \rangle$
(6)	$\langle ., ., ., s_1, a, ., ., ., . \rangle$
(7)	$\langle ., ., ., r_2, ., ., ., ., d \rangle$
(8)	$\langle ., ., ., s_2, ., ., a, ., . \rangle$

FIG. 2.2: Règles de synchronisation pour le système $\langle C, C, S, B, B, B, B \rangle$ où C, S et B correspondent aux automates de la figure 2.1. Les quatre canaux B correspondent respectivement aux sens de communications suivants:
1. $S \to C_1$ 2. $S \to C_2$ 3. $C_1 \to S$ 4. $C_2 \to S$

pour *send*) ou recevant (r) un message. L'état initial est indiqué par « ↘ ». Le serveur possède trois états. Il commence dans l'état central et peut, s'il reçoit un message du premier client (r_1), accéder à l'état 2 d'où il pourra envoyer sa réponse à ce même client (s_1). De façon symétrique, il accède à l'état 3 lorsqu'il reçoit un message du second client. Pour représenter le caractère asynchrone des communications, nous modélisons les canaux par des automates à deux états: « − » lorsque le canal est vide et « × » lorsque le canal a accepté (a) un message mais ne l'a pas encore délivré (d). On utilisera une paire de canaux entre chaque client et le serveur: un par sens de communication. La taille des canaux est ici bornée à 1 message, ce qui est suffisant pour notre système.

Le système peut être décrit comme la synchronisation de 7 instances de ces modèles $\langle C, C, S, B, B, B, B \rangle$ avec les règles de synchronisation de la figure 2.2.

Ces règles listent les transitions qui peuvent être franchies dans le modèle global. Par exemple la première ligne dit que le premier client ne peut franchir la transition s (envoi de message) que si le troisième canal franchit lui aussi sa transition a (acceptation du message sur le canal). Un point indique qu'un composant reste immobile durant cette action.

2.2 Espace d'état

La figure 2.3 page suivante montre le modèle obtenu en synchronisant le serveur avec les deux clients et quatre canaux selon les règles de synchronisation de la figure 2.2 page précédente. Les états du modèle sont étiquetés par des septuplets qui indiquent l'état de l'automate représentant chaque sous-système (dans la figure 2.1 page ci-contre). Par exemple pour l'état q_1, l'étiquette « $211 - - \times -$ » indique que le premier client est dans l'état 2 (attente de réponse), le second client est dans l'état 1 (prêt à envoyer), le serveur est dans l'état 1 (attente de requête), le troisième canal contient un message (celui qui a été envoyé par la premier client au serveur) et les autres canaux sont vides. Sur les $2 \times 2 \times 3 \times 2 \times 2 \times 2 \times 2 = 192$ configurations de sous-automates possibles, seules 15 sont accessibles depuis l'état initial en suivant les règles de franchissement de la section précédente. Seules ces configurations sont représentées.

Puisque nous souhaitons exprimer des propriétés concernant les envois et réceptions de messages, nous pouvons réétiqueter ce modèle en y faisant clairement apparaître la satisfaction des propriétés qui nous intéressent. Nous nous limiterons aux quatre propriétés suivantes:
– r_1: une réponse est en chemin entre le serveur et le premier client
– r_2: une réponse est en chemin entre le serveur et le second client
– d_1: une requête (d pour demande) est en chemin entre le premier client et le serveur
– d_2: une requête est en chemin entre le second client et le serveur
Ces propriétés sont appelées *propositions atomiques*. On notera r_1 lorsque la propriété est vraie et $\bar{r}_1$ lorsqu'elle est fausse. La figure 2.4 page 21, où les étiquettes ont été remplacées par les satisfactions des propositions atomiques, est une *structure de Kripke*.

Comme les systèmes que nous considérons ne s'arrêtent jamais[1], une exécution du modèle est un chemin infini dans cette structure de Kripke. Ici, parce que le modèle comporte plusieurs boucles, il possède un nombre infini d'exécutions possibles.

Le *langage* de cette structure de Kripke est l'ensemble de ses exécutions.

2.3 Formule LTL

Les propositions atomiques r_1, r_2, d_1 et d_2 nous permettent de caractériser des comportements du système en fonction des envois et réceptions de messages.

LTL, la logique temporelle à temps linéaire, permet d'exprimer des contraintes temporelles entre ces propositions atomiques. Par exemple, la propriété « *un client reçoit toujours une réponse du serveur après avoir envoyé une requête* » peut se traduire par la formule LTL suivante:

$$\mathsf{G}(d_1 \rightarrow \mathsf{F}\, r_1) \wedge \mathsf{G}(d_2 \rightarrow \mathsf{F}\, r_2) \tag{2.1}$$

[1] Les systèmes qui possèdent des exécutions finies peuvent être modélisés de façon à boucler sur un état lorsqu'ils ont terminé, afin de faire apparaître des séquences infinies. La section 3.3.7 page 57 détaille cette technique.

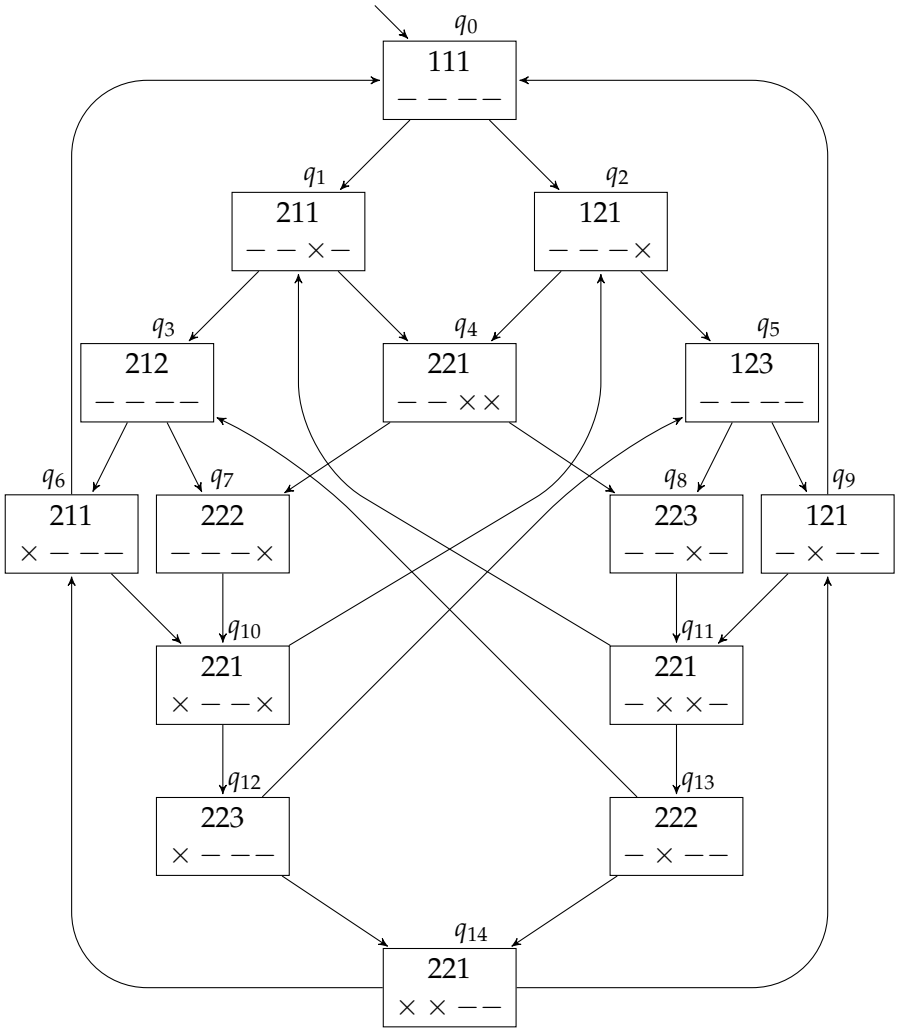


FIG. 2.3: Modèle du système global, après synchronisation des sous-modèles de la figure 2.1 page 18 selon les règles de la figure 2.2.

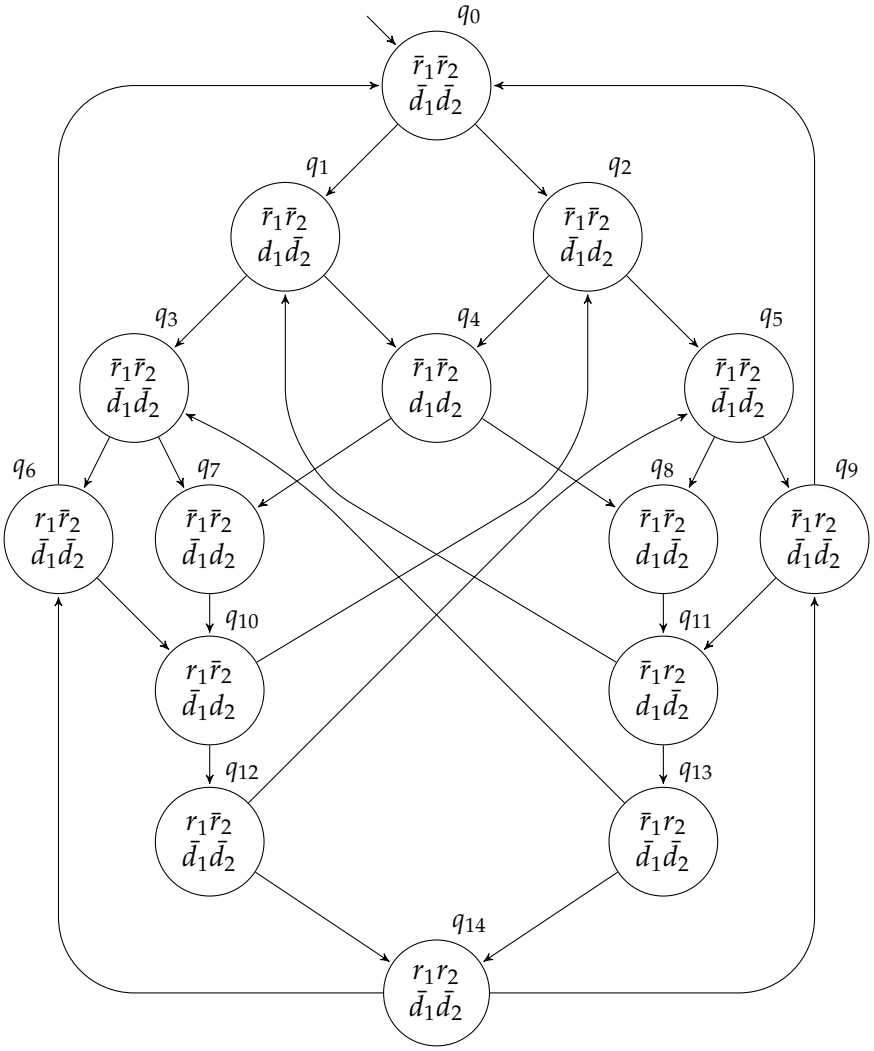


FIG. 2.4: Structure de Kripke du modèle global $\langle C, C, S, B, B, B, B\rangle$.

$\mathsf{G}\,\varphi$ est un opérateur signifiant que φ est vrai à tout instant de l'exécution ; $\mathsf{F}\,\varphi$ dit que φ sera vrai à un instant présent ou futur. Ici, nous disons donc qu'à tout instant, si d_i est vrai, cela implique que r_i sera vrai par la suite.

Pour simplifier les constructions à venir, réduisons la formule à

$$\mathsf{G}(d_1 \rightarrow \mathsf{F}\,r_1) \tag{2.2}$$

Nous pouvons noter que, comme le modèle est symétrique vis-à-vis des deux clients, la formule (2.1) est vraie si et seulement si la formule (2.2) est vraie.

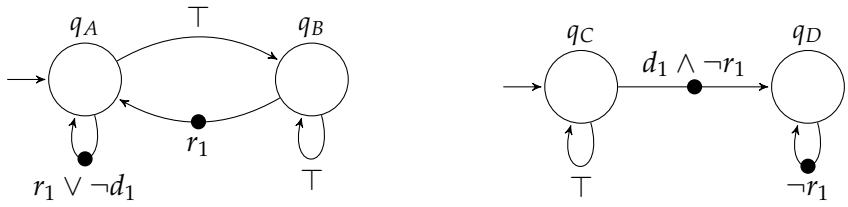
Ces propriétés s'interprètent sur des séquences de valuations des propositions atomiques représentant une exécution du système. Nous dirons que le modèle vérifie la propriété si toutes ses exécutions la vérifient.

Les propriétés que l'on peut exprimer en LTL peuvent être classées en trois types:
- Les *propriétés de sûreté* expriment que *quelque chose de mal ne se produit pas*.
  Une propriété de sûreté interdit certains préfixes d'exécution.
  Pour invalider une telle propriété il suffit de trouver un préfixe fini dans lequel la « mauvaise chose » se produit. Toutes les exécutions que l'on peut construire à partir de ce préfixe invalident la propriété.
  Cette classe de propriétés inclut les *invariants*, qui sont des propriétés devant être vérifiés par chaque état.
- Les *propriétés de vivacité* expriment que *quelque chose de bon se produira*.
  Une propriété de vivacité n'interdit aucun préfixe d'exécution.
  La formule (2.1) en est un exemple: quelque soit le préfixe d'exécution que l'on invente, il est toujours possible de satisfaire la propriété s'il existe un état vérifiant d_1 et d_2 plus loin.
  Pour vérifier une telle propriété il faudrait explorer l'ensemble (infini) de toutes les exécutions (infinies) du modèle.
- Les autres propriétés, qui ne sont ni des propriétés de sûreté ni des propriétés de vivacité peuvent se ramener à la conjonction d'une propriété de sûreté et d'une propriété de vivacité [3, 4].

Le langage associé à une formule LTL est l'ensemble des séquences qui vérifient la formule. En terme de langages on peut donc dire qu'un modèle vérifie une propriété LTL si le langage du modèle est inclus dans le langage de la propriété. Malheureusement cette inclusion d'ensembles ne peut être testée explicitement car ces langages sont de tailles infinies. Il faut donc trouver un moyen d'exprimer ces langages de façon compacte et comparable. L'approche automate du *model checking* classique s'appuie sur les automates de Büchi ; nous utiliserons une généralisation appelée *automates de Büchi généralisés basés sur les transitions*.

2.4 Traduction de la formule en automate

La figure 2.5 page suivante montre deux automates de Büchi basés sur les transitions et acceptant les mêmes langages que la formule (2.1) (figure 2.5a) et sa négation (figure 2.5b).



(a) $A_{\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$ (b) $A_{\neg \mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$

FIG. 2.5: Automates de Büchi basés sur les transitions acceptant les même séquences que les formules LTL $\mathsf{G}(d_1 \to \mathsf{F}\, r_1)$ et $\neg\, \mathsf{G}(d_1 \to \mathsf{F}\, r_1)$.

Ces automates (non-déterministes) s'interprètent de la façon suivante: une séquence infinie de valuations de d_1, d_2, r_1, r_2 est *acceptée* par l'automate si l'on peut trouver dans cet automate un chemin infini

1. étiqueté par des formules propositionnelles compatibles (par exemple la valuation $\bar{d_1} d_2 \bar{r_1} r_2$ est compatible avec la formule $r_1 \vee \neg d_1$ mais pas avec la formule $d_1 \wedge \neg r_1$; toutes les valuations sont compatibles avec $\top$)
2. et qui passe infiniment souvent par des transitions marquées par « ● ».

La marque « ● » est appelée *condition d'acceptation*.

Une définition formelle de ces automates sera donnée dans le chapitre suivant.

Ainsi l'automate $A_{\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$ de la figure 2.5a accepte les séquences infinies qui ne restent pas continûment dans l'état q_B. Il faut pour cela que la séquence valide r_1 infiniment souvent, ou qu'à partir d'un certain point elle valide $\neg d_1$ continûment. L'automate $A_{\neg \mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$ de la figure 2.5b n'accepte que les séquences infinies qui finissent par quitter l'état initial. C'est-à-dire qu'au bout d'un moment une séquence doit vérifier d_1 sans plus jamais vérifier r_1.

Théoriquement, cette traduction est de complexité exponentielle. Il existe des formules LTL de taille n telles que l'automate de Büchi correspondant est de taille $O(2^n)$. En pratique, de très nombreuses formules peuvent être traduites de façon compacte. Comme les propriétés exprimées sont généralement simples, il est rare de voir des automates de plus d'une douzaine d'états. Dans le chapitre 4 nous présenterons différentes façons de traduire une formule LTL en un automate de Büchi, et y apporterons plusieurs optimisations visant à réduire la taille des automates construits.

2.5 Produit et *emptiness check*

Il est possible que le langage accepté par un automate de Büchi soit vide à cause de ses conditions d'acceptation. Par exemple, si l'on retire la condition d'acceptation de la boucle autour de l'état de q_D de la figure 2.5b, il n'existe plus dans cet automate de chemin infini qui traverse « ● » infiniment souvent. Les algorithmes qui permettent de déterminer si le langage associé à un automate est vide ou non sont appelés des *emptiness checks*.

Les formules et conditions d'acceptation des automates de Büchi traditionnellement utilisés en *model checking* portent normalement sur les états et non les transitions. L'intérêt porté aux automates basés sur les transitions que nous utiliserons ici est assez récent. D'autre part, lorsque nous définirons ces automates (définition 27 page 48), nous les équiperons de plusieurs conditions d'acceptation et non d'une seule comme ici. La section 3.3.5 page 51 montre que ces automates sont plus concis que les automates utilisés traditionnellement et permettent d'exprimer certaines classes de propriétés plus facilement.

Les automates de Büchi permettent de réaliser les opérations suivantes sur les langages qu'ils représentent (c'est-à-dire sur les ensembles de séquences infinies de valuations qu'ils acceptent).
– intersection de deux langages (produit synchronisé des automates)
– union de deux langages (somme des automates)
– complément d'un langage (complémentation de l'automate)
– test de vacuité d'un langage (*emptiness check* de l'automate)
Si l'on note $\mathcal{Q}$ l'ensemble des états d'un automate, le produit de deux automates est un automate possédant au pire $|\mathcal{Q}_1| \times |\mathcal{Q}_2|$ états, l'union possède $|\mathcal{Q}_1| + |\mathcal{Q}_2|$ états, la négation $2^{O(|\mathcal{Q}|\log|\mathcal{Q}|)}$ états et l'*emptiness check* peut se faire avec une complexité $O(|\mathcal{Q}|)$.

Une structure de Kripke telle que celle de la figure 2.4 page 21 peut s'interpréter facilement comme un automate de Büchi: il suffit de pousser toutes les étiquettes sur les transitions sortantes de l'état, et de marquer toutes les transitions comme acceptantes (n'importe quel chemin infini est accepté).

Cela suggère une façon de reconnaître si un modèle M vérifie une formule LTL φ, ce qu'on note $M \models \varphi$. Il suffit de vérifier si le langage associé à l'automate A_M représentant le modèle (l'ensemble des exécutions possibles du modèle) est inclut dans le langage associé à l'automate de Büchi A_φ représentant la formule (l'ensemble des séquences de valuations qui valident la formule).

$$M \models \varphi \iff \mathscr{L}(A_M) \subseteq \mathscr{L}(A_\varphi)$$

Cette inclusion peut se réécrire en termes d'intersection ($\cap$), de complément ($\complement$) et de vacuité ($= \emptyset$) de langages:

$$\iff \mathscr{L}(A_M) \cap \complement\mathscr{L}(A_\varphi) = \emptyset$$

Ces opérations peuvent être effectuées sur les automates:

$$\iff \mathscr{L}(A_M) \cap \mathscr{L}(\overline{A_\varphi}) = \emptyset$$
$$\iff \mathscr{L}(A_M \otimes \overline{A_\varphi}) = \emptyset$$

Ainsi tester si M vérifie φ revient à tester la vacuité du produit de A_M avec la négation de A_φ.

Comme la complémentation d'un automate de Büchi est une opération très coûteuse (voir impraticable même pour de petits automates) on préfère nier la formule LTL avant de la traduire. En effet $\mathscr{L}(\overline{A_\varphi}) = \mathscr{L}(A_{\neg\varphi})$. L'équivalence sur laquelle est fondée l'approche automate du *model checking* Vardi [136, 139] s'exprime alors ainsi:

$$M \models \varphi \iff \mathscr{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$$

La figure 2.6 page suivante montre le produit de la structure de Kripke de la figure 2.4 page 21 (interprétée comme un automate de Büchi) avec l'automate $A_{\neg\,\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$ de la figure 2.5b. Pour alléger la figure, les formules propositionnelles qui devraient étiqueter les arcs ont été omises. L'ensemble des états du nouvel automate est le produit cartésien des deux automates. Nous avons donc ici $15 \times 2 = 30$ états: les 15 états à gauche de la figure (une fois la page ou la tête tournée) correspondent à la synchronisation de la structure de Kripke avec l'état q_C de l'automate $A_{\neg\,\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}$, les 15 états de droite correspondent à l'état de q_D. Comme la boucle autour de ce dernier état était étiquetée par $\neg r_1$, les états du produit correspondant à des états étiquetés par r_1 dans la structure de Kripke n'ont pas de successeur. Enfin huit transitions relient les états de gauche aux états de droite et correspondent à la transition au centre de la figure 2.5b, ils quittent donc uniquement des états étiquetés par $\bar{r}_1 d_1$ dans la structure de Kripke .

Dans cet automate produit, un seul état n'est pas accessibles depuis l'état initial. Dans une construction *à la volée* où seules les portions explorées du produit synchronisé sont construites, cet état inaccessible ne serait jamais construit.

Les conditions d'acceptation de l'automate représentant la formule ont été préservées dans l'automate produit. Pour répondre à la question

$$\mathscr{L}(A_M \otimes A_{\neg\,\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}) \stackrel{?}{=} \emptyset$$

nous devons donc déterminer s'il existe un chemin acceptant dans cet automate, c'est à dire un chemin infini qui passe infiniment souvent par « ● ». (Les propositions atomiques n'importent pas.)

Comme l'indiquent les transitions tracées en gras sur la figure 2.6, un tel chemin existe (il y en a d'autres que celui-ci). Dans le chapitre 5 nous présenterons les différents algorithmes d'*emptiness check* existant pour résoudre ce problème. Nous introduirons des optimisations et des heuristiques pour améliorer leurs performances.

Puisque $\mathscr{L}(A_M \otimes A_{\neg\,\mathsf{G}(d_1 \to \mathsf{F}\, r_1)}) \neq \emptyset$, on en déduit que $M \not\models \mathsf{G}(d_1 \to \mathsf{F}\, r_1)$. La séquence d'exécution en gras représente un contre-exemple de la formule $\mathsf{G}(d_1 \to \mathsf{F}\, r_1)$. En la projetant sur l'espace d'état du modèle (figure 2.3 page 20) on peut comprendre le comportement du système qui invalide cette formule:

1. le client C_1 envoie une requête ;
2. le client C_2 envoie une requête ;
3. le serveur reçoit la requête du client C_2 ;
4. le serveur envoie une réponse au client C_2 ;

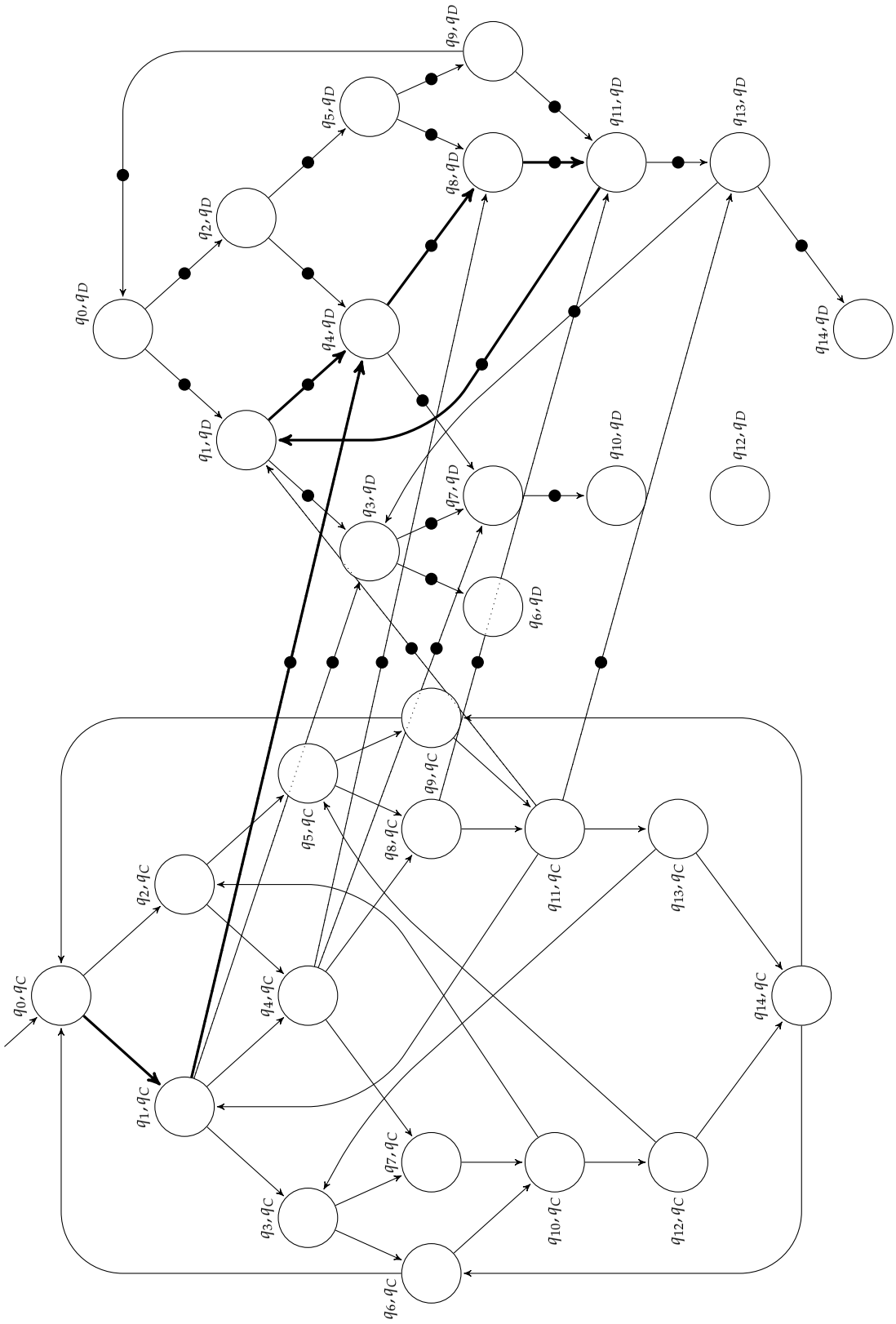


FIG. 2.6: Produit de la structure de Kripke de la figure 2.4 page 21 (interprétée comme un automate de Büchi) avec l'automate $A_{\neg \mathsf{G}(d_1 \rightarrow \mathsf{F} r_1)}$ de la figure 2.5b.

5. le client C_2 reçoit la réponse du serveur ;
6. le scénario se répète à partir du point 2.

Autrement dit ce modèle permet au message envoyé par le client C_1 de ne pas être délivré au serveur tant que le client C_2 envoie des messages.

2.6 Hypothèses d'équité

Il y a peu de chance que le contre-exemple précédent corresponde à une exécution réaliste du système. Dans un système où les communications sont sans perte et asynchrones, les messages peuvent éventuellement subir un retard, mais ils finissent par être délivrés.

Nous pourrions modifier le modèle pour le rendre un peu plus fidèle au système, en modélisant de façon plus fine le système de communication. Cette modification est complexe.

Une autre façon de résoudre ce problème est d'enrichir notre technique de vérification pour prendre en compte des *hypothèses d'équité*. Ces hypothèses portent sur le modèle et permettent de restreindre l'ensemble des exécutions qui doivent être considérées lors de la vérification.

Dans le cas présent, nous souhaiterions que les transitions correspondant aux règles (5) et (7) de la figure 2.2 page 18 ne puissent être franchissables infiniment souvent sans jamais être franchies.

Dans notre contre-exemple, chaque fois que le serveur reçoit le message du client C_2, le système se trouve dans un état où la transition correspondant à la réception du message du client C_1 est franchissable. Nous voulons donc que cette transition finisse par être franchie.

La formule $\mathsf{G}(d_1 \rightarrow \mathsf{F}\, r_1)$ est vérifiée par le modèle sous cette hypothèse.

Il s'agit ici d'une hypothèse d'équité *forte*. Différents types d'hypothèses d'équité seront présentés dans le chapitre 7. Nous introduirons alors les automates de Streett, qui permettent de prendre en compte ces hypothèses d'équité forte directement dans la représentation du modèle, et proposerons un algorithme d'*emptiness check* original dédié à ces automates.

2.7 Explosion combinatoire

Le *model checking* par l'approche automate souffre du problème de l'explosion de l'espace d'état qu'il doit traiter [134].

Sur notre exemple, nous sommes partis d'un modèle simpliste décrit de façon compacte par les figures 2.1 et 2.2 page 18 pour obtenir l'automate à 15 états et 28 transitions de la figure 2.4 page 21. Après un produit synchronisé avec l'automate représentant la négation de la formule à vérifier, l'automate qu'il fallait finalement explorer pour y chercher un chemin acceptant (figure 2.6 page précédente) possédait 29 états (accessibles) et 52

transitions. L'explosion combinatoire est ici très faible et peu représentative (l'exemple a l'avantage de pouvoir être présenté entièrement sur une page).

Le même système avec un client supplémentaire posséderait 78 états et 208 transitions avant le produit synchronisé, puis 120 états (accessibles) et 364 transitions après le produit. La taille de l'espace d'état croît de façon exponentielle avec le nombre de processus qui peuvent s'exécuter en parallèle, puisqu'il faut prendre en compte tous les entrelacements d'exécutions possibles.

Il est important de lutter contre cette explosion combinatoire en cherchant à réduire tout ce qui peut l'être: la taille de l'automate représentant la formule (chapitre 4), mais aussi et surtout la taille de la structure de Kripke représentant le modèle. Même si nous n'allons pas les aborder (autrement que par un exemple dans le chapitre 6), il faut être conscient que diverses techniques peuvent être utilisées pour réduire la taille de l'automate A_M. Par exemple une étude des symétries du modèle peut permettre de « replier » l'automate A_M [29, 2, 81, 126]. Les *stubborn sets* [132, 142], *persistent sets* [64] et *ample sets* [101] sont des techniques dites d'*ordre partiel* dans lesquelles des séquences d'exécution équivalentes à des permutations de transitions près sont identifiées. Les graphes de dépliage proposent une autre approche pour la représentation des ordres partiels [38, 48].

2.8 Résumé de l'approche automate

La figure 2.7 page ci-contre résume les grandes lignes de l'approche automate du *model checking*. Les rectangles aux coins carrés y représentent des données, tandis que ceux aux coins arrondis sont des algorithmes.

En haut, le modèle et la formule LTL sont les deux éléments fournis par la personne qui souhaite déterminer si le modèle M vérifie la formule φ. Le résultat, tout en bas, est soit l'assurance que le modèle vérifie la formule ($M \models \varphi$), soit un contre-exemple (une exécution du modèle qui invalide la formule).

Comme nous l'avons vu, le modèle de haut niveau est d'abord traduit sous la forme d'un automate A_M dont le langage représente l'ensemble des exécutions possibles du modèle. La formule LTL φ est elle aussi traduite en un automate de Büchi $A_{\neg\varphi}$ représentant l'ensemble des exécutions qui invalident la formule (ou qui valident la négation de la formule).

Le produit synchronisé de ces deux automates est un automate de Büchi qui reconnaît l'intersection de ces deux langages, c'est-à-dire l'ensemble des exécutions du modèle qui invalident la formule.

Un algorithme d'*emptiness check* parcourt enfin le produit pour déterminer si le langage accepté par l'automate est vide ou non.

Le produit synchronisé peut être construit à la volée au fur et à mesure que l'algorithme d'*emptiness check* explore le produit ; cela évite d'avoir à construire un énorme produit si l'*emptiness check* n'a pas besoin de tout parcourir pour répondre. L'espace d'état et l'automate de la formule peuvent eux aussi être construits à la volée pour répondre au besoin du produit synchronisé: la synchronisation peut « couper » des zones de l'un ou l'autre des automates qu'il est inutile d'avoir générées.

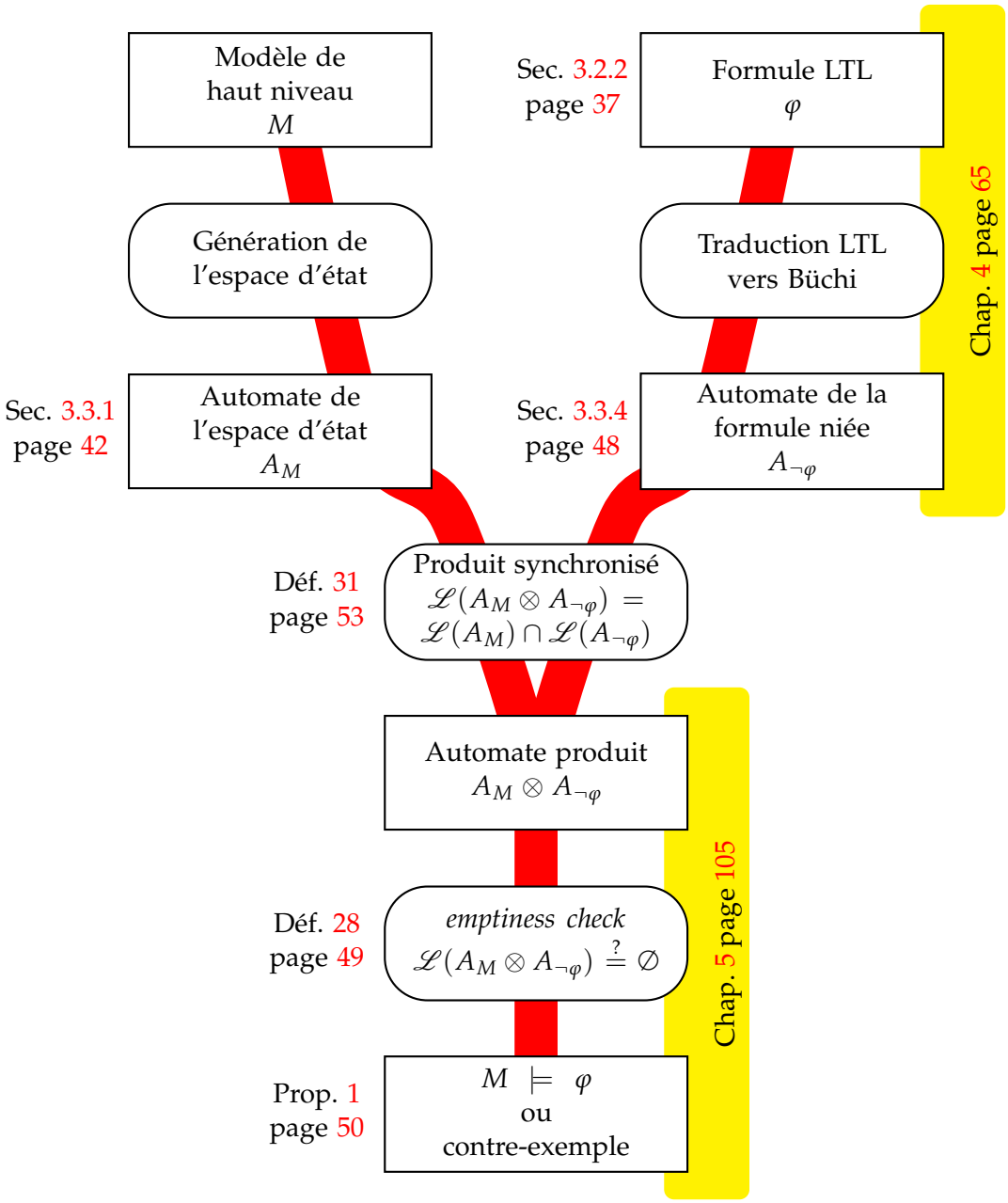


FIG. 2.7: Approche automate du *model checking*.

Chapitre 3

Automates de Büchi basés sur les transitions

Ce chapitre a failli s'appeler « *Définitions préliminaires* » car il a un double objectif. Tout d'abord il définit précisément les formalismes et notations que nous utiliserons dans tous les chapitres suivants. Les *automates de Büchi généralisés basés sur les transitions* (TGBA) nous permettront de représenter les systèmes, alors que nous utiliserons les formules de *logique temporelle à temps linéaire* (LTL) pour spécifier les propriétés à vérifier. Ces deux formalismes permettent d'exprimer des ensembles séquences de configurations du système, nous commencerons donc par définir ces séquences.

D'autre part il motive le choix des TGBA par rapport aux autres types d'automates de Büchi plus traditionnellement employés. Nous devrons donc aussi définir ces derniers afin de pouvoir établir des parallèles.

3.1 Séquences et propositions atomiques

3.1.1 Propositions atomiques

Nous étiquetterons les états du système à vérifier, ou selon les cas ses transitions, par la valuation d'un ensemble fini de variables propositionnelles. Par exemple l'état d'un feu tricolore peut être symbolisé par la valuation de trois variables propositionnelles r (rouge), o (orange) et v (vert) indiquant chacune si le feu correspondant est allumé.

À l'extrême, lorsque la configuration d'un système est caractérisée par l'état de sa mémoire, on peut associer une variable propositionnelle à chaque bit de la mémoire. Une valuation de l'ensemble de ces variables correspondrait alors à un instantané de l'état du système. Dans la pratique ces variables sont associées à des propriétés de niveau un peu plus haut (que le bit!) que l'on souhaite observer sur le modèle.

Ces variables propositionnelles seront appelées *propositions atomiques*. Leur ensemble est traditionnellement noté AP (pour *atomic propositions*).

Nous appelons *littéral* une proposition atomique ou sa négation. p, $\neg p$, q et $\neg q$ sont les littéraux de $AP = \{p, q\}$. Un *cube* est une conjonction de littéraux, par exemple $\neg p$ et $p \wedge q$ sont des cubes. Un *minterm* est un cube dans lequel toutes les propositions atomiques apparaissent.

Les valuations possibles de l'ensemble des propositions atomiques peuvent être exprimées sous la forme de *minterms*. Par exemple si $AP = \{p, q\}$, le système peut prendre au plus quatre états: $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, $\neg p \wedge \neg q$.

La notation 2^{AP} désigne l'ensemble des parties de AP. Avec la dernière définition de AP, nous avons $2^{AP} = \{\{p, q\}, \{p\}, \{q\}, \emptyset\}$.

Il existe une bijection entre 2^{AP} et l'ensemble des *minterms* sur AP: il suffit d'interpréter les éléments de 2^{AP} comme des listes de propositions positives, toutes les autres propositions étant niées.

$$\begin{array}{rcll} \{p, q\} & \leftrightarrow & p \wedge q & \text{(ou encore } pq\text{)} \\ \{p\} & \leftrightarrow & p \wedge \neg q & (p\bar{q}) \\ \{q\} & \leftrightarrow & \neg p \wedge q & (\bar{p}q) \\ \emptyset & \leftrightarrow & \neg p \wedge \neg q & (\bar{p}\bar{q}) \end{array}$$

Nous utiliserons l'une ou l'autre de ces représentations de façon équivalente et nous noterons 2^{AP} l'ensemble des *minterms* des propositions atomiques AP.

De façon similaire, nous désignerons par $2^{2^{AP}}$ l'ensemble des formules propositionnelles sur AP, car il existe une relation entre ces deux ensembles. Chaque élément de $2^{2^{AP}}$ peut en effet être interprété comme une disjonction de *minterms*. Par exemple $\{\{p\}, \emptyset\}$ s'interprète comme $(p \wedge \neg q) \vee (\neg p \wedge \neg q)$. Réciproquement, toute formule propositionnelle peut être exprimée sous cette forme: nous pouvons identifier la formule $\neg q$ avec l'ensemble $\{\{p\}, \emptyset\}$.

3.1.2 Séquences

Définition 1. *Nous noterons $\mathbb{N} = \{0, 1, 2, \ldots\}$ l'ensemble des entiers naturels, et $\omega \notin \mathbb{N}$ le premier ordinal infini. La relation d'ordre définie sur les entiers naturels est étendue à $\mathbb{N} \cup \{\omega\}$ avec $n < \omega$ pour tout $n \in \mathbb{N}$. On étend de la même façon l'addition et la soustraction entières avec $\omega + n = \omega - n = \omega + \omega = \omega$ pour tout $n \in \mathbb{N}$.*

Définition 2. *Nous utiliserons les notations suivantes pour désigner des intervalles de $\mathbb{N} \cup \{\omega\}$ compris entre $n \in \mathbb{N}$ et $m \in \mathbb{N} \cup \{\omega\}$:*

$$\begin{aligned} [\![n, m]\!] &= \{i \in \mathbb{N} \cup \{\omega\} \mid n \leq i \leq m\} \\ [\![n, m[\![&= \{i \in \mathbb{N} \mid n \leq i < m\} \\ ]\!]n, m]\!] &= \{i \in \mathbb{N} \cup \{\omega\} \mid n < i \leq m\} \\ ]\!]n, m[\![&= \{i \in \mathbb{N} \mid n < i < m\} \end{aligned}$$

Notons que $[\![0, \omega[\![= \mathbb{N}$ et $[\![n, n[\![=]\!]n, n]\!] =]\!]n, n[\![= \emptyset$.

Définition 3. *Soit Σ un ensemble non vide et $n \in \mathbb{N} \cup \{\omega\}$ un ordinal. Une* séquence *de longueur n sur Σ est une fonction $\sigma : [\![0, n[\![\mapsto \Sigma$ faisant correspondre un ensemble d'indices $[\![0, n[\![$ à des éléments de Σ.*

Définition 4. *Pour un $n \in \mathbb{N} \cup \{\omega\}$, on notera Σ^n l'ensemble des séquences de longueur n sur Σ. Σ^ω désigne ainsi l'ensemble des séquences infinies d'éléments de Σ. Σ est assimilé de façon triviale à l'ensemble Σ^1 des séquences de longueur 1. Enfin, Σ^0 contient l'unique fonction « nullaire » notée $\varepsilon_\Sigma : \emptyset \mapsto \Sigma$, qui représente la séquence vide.*

Définition 5. *L'ensemble des séquences finies sera noté $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$. L'ensemble des séquences finies ou infinies sera noté $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.*

Définition 6. *On notera $|\sigma|$ la longueur d'une séquence $\sigma \in \Sigma^\infty$. C'est-à-dire que pour $n \in \mathbb{N} \cup \{\omega\}$ on a $|\sigma| = n \iff \sigma \in \Sigma^n$.*

Nous appellerons parfois une séquence un *mot*, surtout dans les parties qui s'appuient sur des automates. Il n'y a pas d'autre différence que celle du vocabulaire. L'ensemble Σ est alors appelé *alphabet* et ses éléments sont des *lettres*. Un n-mot est un mot de longueur n, et ε_Σ est appelé *mot vide*.

Définition 7. *Soient $n \in \mathbb{N} \cup \{\omega\}$ un ordinal, $\sigma \in \Sigma^n$ une séquence de taille n et $i \in [\![0, n]\!]$ un indice de cette séquence. Le suffixe de σ commençant à la position i est la séquence notée $\sigma^i : [\![0, n-i[\![\mapsto \Sigma$ et définie par $\forall j \in [\![0, n-i[\![, \sigma^i(j) = \sigma(i+j)$.*

Notons qu'un suffixe est de longueur infinie si et seulement si sa séquence d'origine l'est aussi ($|\sigma^i| = \omega \iff |\sigma| = \omega$).

Définition 8. *Soient $(n_1, n_2) \in (\mathbb{N} \cup \{\omega\})^2$ deux ordinaux, et $\sigma_1 \in \Sigma^{n_1}$ et $\sigma_2 \in \Sigma^{n_2}$ deux séquences de tailles correspondantes.*

La concaténation *de ces deux séquences est la séquence $\sigma_3 \in \Sigma^{n_1+n_2}$ notée $\sigma_3 = \sigma_1 \cdot \sigma_2$ et définie par:*

$$\sigma_3(i) = \begin{cases} \sigma_1(i) & \text{si } i < n_1 \\ \sigma_2(i - n_1) & \text{sinon} \end{cases}$$

À noter que si $n_1 = \omega$, alors $\sigma_3 = \sigma_1$ par définition.

Puisque nous avons assimilé les séquences de longueur 1 avec leur unique valeur, une séquence sera souvent représentée par la concaténation de ses valeurs: $\sigma = \sigma(0) \cdot \sigma(1) \cdot \sigma(2) \cdots$.

Exemple 1. *Soit $\Sigma = \{a, b\}$. La séquence de Σ^ω définie par*

$$\sigma(i) = \begin{cases} a & \text{si } i \text{ est pair} \\ b & \text{sinon} \end{cases}$$

peut être représentée par $\sigma = a \cdot b \cdot a \cdot b \cdot a \cdot b \cdots$.

3.1.3 Expressions ω-rationnelles

Définition 9. *Soit Σ un alphabet. Un langage sur Σ est un ensemble (fini ou non) de mots sur Σ.*

Nous définissons maintenant les expressions ω-rationnelles, qui permettent d'écrire et de composer des langages de façon plus concise que la notation ensembliste (surtout lorsque ces ensembles sont de taille infinie).

Définition 10. *L'ensemble des expressions ω-rationnelles sur un alphabet Σ est défini par induction de la façon suivante:*
- *un mot $m \in \Sigma^\infty$ est une expression ω-rationnelle ;*
- *si w_1 et w_2 sont deux expressions ω-rationnelles, alors w_1^ω, w_1^*, $(w_1 + w_2)$ et $(w_1 \cdot w_2)$ sont des expressions ω-rationnelles.*

Le langage d'une expression ω-rationnelle w, noté $\mathscr{L}(w)$, est défini par

$$\begin{aligned}
\mathscr{L}(m) &= \{m\} \\
\mathscr{L}\big((w_1 + w_2)\big) &= \mathscr{L}(w_1) \cup \mathscr{L}(w_2) \\
\mathscr{L}\big((w_1 \cdot w_2)\big) &= \{m_1 \cdot m_2 \mid m_1 \in \mathscr{L}(w_1), m_2 \in \mathscr{L}(w_2)\} \\
\mathscr{L}(w_1^*) &= \{\varepsilon_\Sigma\} \cup \bigcup_{n \in \mathbb{N}} \{m_0 \cdot m_1 \cdots m_n \mid m_i \in \mathscr{L}(w_1) \text{ pour tout } i \leq n\} \\
\mathscr{L}(w_1^\omega) &= \{m_0 \cdot m_1 \cdots \mid m_i \in \mathscr{L}(w_1) \text{ pour tout } i \in \mathbb{N}\}
\end{aligned}$$

Pour supprimer des parenthèses et alléger les notations, nous supposerons les priorités classiques de ces opérateurs: * et $^\omega$ sont prioritaires sur $\cdot$ qui est prioritaire sur $+$.

Exemple 2. *Prenons $\Sigma = \{a, b, c\}$. Le langage associé à l'expression $(a^* \cdot b)^\omega$ est l'ensemble des mots de Σ^ω où b apparaît infiniment souvent et c jamais. Le langage de $(a + b)^\omega$ est l'ensemble des mots de Σ^ω où c n'apparaît pas. L'expression $(a^\omega \cdot b)^\omega$ est équivalente à a^ω (car $a^\omega \cdot b$ est équivalente à a^ω), alors que $(a^\omega \cdot b)^*$ est équivalente à $\varepsilon_\Sigma + a^\omega$.*

Notons que les expressions ω-rationnelles ne fournissent pas de mécanisme pour exprimer le complément d'un langage ou l'intersection de deux langages. Les langages qui résultent de ces opérations peuvent cependant être exprimés par des expressions ω-rationnelles [23].

3.1.4 Séquences infinies sur 2^{AP}

Par la suite, nous nous intéresserons souvent à des mots infinis utilisant l'alphabet $\Sigma = 2^{AP}$. Comme nous l'avons vu section 3.1.1 page 31, AP désigne un ensemble de propositions atomiques dont la valuation caractérise la configuration d'un système. Un élément de 2^{AP} correspond donc à une configuration, et une séquence sur 2^{AP} représente l'évolution du système au cours du temps.

Exemple 3. *Revenons sur la modélisation du feu tricolore mentionnée page 31, qui utilise trois variables propositionnelles r, o et v pour indiquer l'état des trois feux. Si* $AP = \{r, o, v\}$, 2^{AP} *peut être représenté par* $\Sigma = \{rov, ro\bar{v}, r\bar{o}v, r\bar{o}\bar{v}, \bar{r}ov, \bar{r}o\bar{v}, \bar{r}\bar{o}v, \bar{r}\bar{o}\bar{v}\}$.

Le langage correspondant à l'expression ω-rationnelle $(\bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v})^\omega$ *est réduit à l'unique séquence infinie* $\bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v} \cdot \bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v} \cdot \bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v} \cdots$. *Cette séquence décrit un feu tricolore dont tous les feux sont éteints, sauf le feu orange qui n'est allumé qu'aux instants pairs.*

Lorsque le langage correspondant à une expression ω-rationnelle se réduit à une seule séquence comme dans l'exemple précédent, on assimilera l'expression ω-rationnelle à cette séquence. On se permettra donc d'écrire *la séquence* $(\bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v})^\omega$.

Définition 11. *Soient AP et* AP' *deux ensembles de propositions atomiques tels que* $AP \supseteq AP'$, *et une séquence* $\sigma \in (2^{AP})^\omega$.

La projection de σ *sur* AP' *est la séquence notée* $\sigma_{|AP'}$ *telle que* $\forall i \geq 0$, $\sigma_{|AP'}(i) = \sigma(i) \cap AP'$.

Pour un ensemble $L \subseteq (2^{AP})^\omega$, *on notera aussi* $L_{|AP'} = \{\sigma_{|AP'} \mid \sigma \in L\}$.

Exemple 4. $(\bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v})^\omega_{|\{r,v\}} = (\bar{r}\bar{v})^\omega$.

Définition 12. *Soient AP et* AP' *deux ensembles de propositions atomiques disjoints, et deux séquences* $\sigma \in (2^{AP})^\omega$ *et* $\sigma' \in (2^{AP'})^\omega$. *La composition de* σ *et* σ' *est la séquence notée* $\pi = \sigma \parallel \sigma'$ *et telle que* $\forall i \geq 0$, $\pi(i) = \sigma(i) \cup \sigma(i')$.

Exemple 5. $(a\bar{b})^\omega \parallel (c \cdot \bar{c})^\omega = (a\bar{b}c \cdot a\bar{b}\bar{c})^\omega$

3.1.5 Formalismes de représentation de langages

La figure 3.1 page suivante regroupe différents formalismes pouvant être utilisés pour représenter des langages, c'est-à-dire, en ce qui nous concerne, des sous-ensembles de Σ^ω.

Naturellement, les expressions ω-rationnelles ne permettent pas d'exprimer tout sous-ensemble de Σ^ω. Par exemple si $\Sigma = \{a, b\}$, le langage constitué du seul mot

$$a \cdot b \cdot \underbrace{a \cdot a}_{2 \text{ fois}} \cdot \underbrace{b \cdot b}_{2 \text{ fois}} \cdots \underbrace{a \cdot a \cdots a}_{i \text{ fois}} \cdot \underbrace{b \cdot b \cdots b}_{i \text{ fois}} \cdots$$

n'est pas ω-rationnel.

Si Σ est interprété comme une valuation de propositions atomiques, des ensembles de séquences de Σ^ω peuvent être représentés par des formules de logique temporelle. Les logiques F1S, S1S et LTL sont introduites dans la section suivante. Par la suite nous utiliserons essentiellement LTL ; F1S et S1S ne sont présentées que pour aider le lecteur peu familier avec LTL à situer son pouvoir d'expression.

Ces ensembles peuvent aussi être représentés par des automates reconnaissant des mots infinis: des ω-automates. Les automates de Büchi et leurs généralisations (GBA et TGBA) seront présentés section 3.3 page 42. Dans ce mémoire nous utiliserons principalement les TGBA, mais nous aurons besoin de nous comparer aux deux autres formalismes (automates de Büchi classiques et GBA) qui sont plus couramment utilisés dans la littérature.

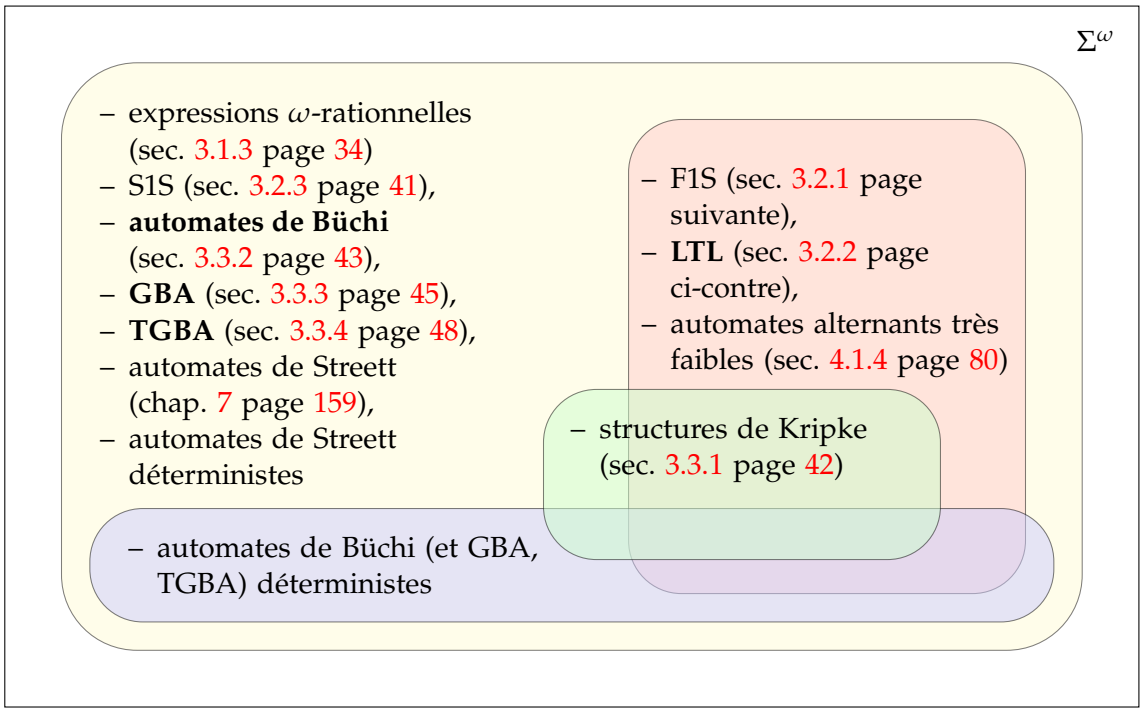


FIG. 3.1: Pouvoir d'expression des différents formalismes introduits.

Enfin les automates de Streett et les automates alternants ne seront employés que localement. Nous les définirons hors de ce chapitre, à l'occasion.

3.2 Logiques temporelles

Revenons à nouveau sur la modélisation du feu tricolore avec ses trois variables propositionnelles r, o et v pour indiquer l'état des trois feux.

Ces propositions atomiques et les connecteurs classiques de la logique propositionnelle ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$) nous permettent de caractériser l'état du feu tricolore à un instant donné. Par exemple $(\neg r) \wedge (\neg o) \wedge v$ désigne l'état où le feu vert est allumé tandis que les deux autres sont éteints.

Un tel formalisme (la logique propositionnelle) ne nous permet pas d'exprimer des propriétés telles que *le feu ne reste pas continûment rouge*, ou encore *le feu orange clignote*, qui nécessitent de caractériser l'évolution du système par rapport au temps.

Dans cette section nous introduisons trois logiques permettant d'exprimer des propriétés où l'état du système est fonction du temps [129]. Les deux premières, la *logique monadique du premier ordre à un successeur* (F1S) et la *logique temporelle à temps linéaire* (LTL) sont aussi expressives l'une que l'autre. Nous utiliserons uniquement LTL par la suite. Comme son

nom le laisse supposer la *logique monadique du second ordre à un successeur* (S1S) est plus expressive que F1S.

3.2.1 F1S: Logique monadique du premier ordre à un successeur

En F1S, les propositions atomiques deviennent des prédicats unaires, paramétrés par le temps: $r(t)$, $o(t)$ et $v(t)$ indiquent ainsi si le feu correspondant est allumé à l'instant t, le temps étant discret.

Les formules F1S sont construites à partir de
– 0 : un symbole désignant l'instant initial
– $t+1$: un opérateur retournant l'instant successeur immédiat
– $t \leq u$: une relation d'ordre total sur les instants
– $\exists t, \forall t$: des quantificateurs du premier ordre
– $AP = \{r, o, v, \ldots\}$: un ensemble de prédicats ($r(t)$, $o(t)$, $v(t)$ indiquant ici si les feux sont allumés à l'instant t)
– $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$: les opérateurs usuels de la logique propositionnelle
Les prédicats d'une formule F1S φ s'interprètent à partir d'une séquence $\sigma \in (2^{AP})^\omega$. Le prédicat $x(t)$ est vrai si et seulement si $x \in \sigma(t)$. On dira que la séquence σ satisfait φ, noté $\sigma \models \varphi$, si la formule est correcte en y remplaçant $x(t)$ par $x \in \sigma(t)$ pour chaque prédicat x.

Pour un ensemble de propriétés AP donné, nous noterons $\mathscr{L}_{AP}(\varphi)$ l'ensemble des séquences infinies de 2^{AP} qui satisfont φ (on les appellera aussi des *modèles* de φ):

$$\mathscr{L}_{AP}(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$$

Exemple 6. *Voici trois propriétés et leur expression en F1S.*

1. *Le feu ne reste pas continûment* rouge.
   $\neg\forall t.(r(t) \wedge \neg o(t) \wedge \neg v(t))$
2. *Toute configuration* orange *est immédiatement suivie de* rouge.
   $\forall t.((\neg r(t) \wedge o(t) \wedge \neg v(t)) \rightarrow (r(t+1) \wedge \neg o(t+1) \wedge \neg v(t+1)))$
3. *Le système passe infiniment souvent par la configuration* vert.
   $\forall t.\exists u.(t \leq u) \wedge (\neg r(u) \wedge \neg o(u) \wedge v(u))$

La séquence $(\bar{r}o\bar{v} \cdot \bar{r}\bar{o}\bar{v})^\omega$ de l'exemple 3 page 34 ne satisfait que la première propriété de l'exemple 6.

3.2.2 LTL: Logique temporelle à temps linéaire

Syntaxe *Une formule LTL* est construite à partir des éléments suivants:
– Un ensemble $AP = \{p_1, p_2, \ldots\}$ de propositions atomiques,
– des connecteurs booléens $\wedge$ et $\neg$,
– des opérateurs temporels X (*next*) et U (*until*).

Définition 13. *Soit AP un ensemble de propositions atomiques. L'ensemble* LTL_{AP} *des formules LTL sur AP est le plus petit ensemble tel que*

– *les propositions atomiques sont des formules LTL ($AP \subseteq \mathrm{LTL}_{AP}$)*
– *si f_1 et f_2 sont deux formules LTL, alors $(f_1 \wedge f_2)$, $\neg f_1$, $\mathsf{X} f_1$ et $(f_1 \mathsf{U} f_2)$ sont aussi des formules LTL.*

Bien entendu nous nous dispenserons des parenthèses lorsqu'elles ne s'avèrent pas indispensables.

Sémantique. Une formule de LTL_{AP} s'interprète sur une séquence infinie $\sigma \in (2^{AP})^\omega$.

Définition 14. *Pour toute proposition atomique p_i et toutes formules LTL f_1 et f_2, la satisfaction d'une formule LTL f par rapport à $\sigma \in (2^{AP})^\omega$ est notée $\sigma \models f$ et définie inductivement de la façon suivante:*

$$
\begin{array}{ll}
\sigma \models p & \textit{ssi } p \in \sigma(0) \\
\sigma \models \neg f_1 & \textit{ssi } \neg(\sigma \models f_1) \\
\sigma \models f_1 \wedge f_2 & \textit{ssi } \sigma \models f_1 \textit{ et } \sigma \models f_2 \\
\sigma \models \mathsf{X} f_1 & \textit{ssi } \sigma^1 \models f_1 \\
\sigma \models f_1 \mathsf{U} f_2 & \textit{ssi } \exists i \geq 0 \textit{ tel que } \sigma^i \models f_2 \textit{ et } \forall j \in [\![0, i-1]\!],\ \sigma^j \models f_1
\end{array}
$$

Nous étendrons cette sémantique aux séquences finies dans la définition 36 de la section 3.3.7 page 57. Dans tout le reste du document nous ne considèrerons que des séquences infinies.

Définition 15. *Soit AP un ensemble de propositions atomiques et $\varphi \in \mathrm{LTL}_{AP}$ une formule LTL. Le langage de la formule φ est l'ensemble des séquence infinies sur 2^{AP} qui satisfont φ.*

$$\mathscr{L}_{AP}(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$$

Lien avec F1S. En LTL, la valeur des propositions atomiques varie au cours du temps (toujours discret), mais ce temps n'apparaît pas explicitement dans la formule: il est manipulé par les opérateurs X et U. Par exemple la formule p_1 indique que la propriété correspondant à la proposition atomique p_1 est vraie à l'instant présent, et la formule $\mathsf{X} p_1$ indique que cette propriété est vraie à l'instant suivant. Les expressions F1S correspondantes seraient $p_1(0)$ et $p_1(0+1)$.

Une façon de donner la sémantique LTL est de fournir une traduction vers F1S. Notons $p_1, p_2, \ldots$ les propositions atomiques de LTL, et $p_1(t), p_2(t), \ldots$ les prédicats correspondants en F1S. Une formule LTL f correspond à l'expression F1S $[f]_0$ définie inductivement de la façon suivante. Si p_i est une proposition atomique, et f_1 et f_2 des formules

LTL:

$$\begin{aligned}
[p_i]_t &= p_i(t) \\
[\neg f_1]_t &= \neg [f_1]_t \\
[f_1 \wedge f_2]_t &= [f_1]_t \wedge [f_2]_t \\
[\mathsf{X}\, f_1]_t &= [f_1]_{t+1} \\
[f_1 \,\mathsf{U}\, f_2]_t &= \exists u.((\forall v.((t \leq v) \wedge (v+1 \leq u)) \rightarrow [f_1]_v) \wedge [f_2]_u)
\end{aligned}$$

(Le nom des variables u et v étant bien entendu choisi de façon unique dans le cas où plusieurs U sont imbriqués.)

Une séquence $\sigma \in (2^{AP})^\omega$ satisfait une formule LTL f, noté $\sigma \models f$, si et seulement si elle satisfait l'expression F1S correspondante $\sigma \models [f]_0$.

Opérateurs dérivés. Les autres opérateurs booléens usuels ($\vee$, $\leftrightarrow$, $\rightarrow$, ...) peuvent être définis comme abréviations de façon classique à partir de $\wedge$ et $\neg$. $\top$ (vrai) et $\bot$ (faux) se définissent comme $\top = p \vee \neg p$ et $\bot = p \wedge \neg p$.

D'autres opérateurs temporels courants tels que F (*eventually*), G (*always*) et R (*release*) sont aussi définis en tant qu'abréviations:

$$\begin{aligned}
\mathsf{F}\, f_1 &= \top \,\mathsf{U}\, f_1 \\
f_1 \,\mathsf{R}\, f_2 &= \neg(\neg f_1 \,\mathsf{U}\, \neg f_2) \\
\mathsf{G}\, f_1 &= \neg\, \mathsf{F}\, \neg f_1 = \neg(\top \,\mathsf{U}\, \neg f_1) = \bot \,\mathsf{R}\, f_1
\end{aligned}$$

Exemple 7. *Voici une façon d'exprimer en LTL les trois propriétés de l'exemple 6 page 37.*

1. *Le feu ne reste pas continûment* rouge.
   $\neg\, \mathsf{G}(r \wedge \neg o \wedge \neg v)$
2. *Toute configuration* orange *est immédiatement suivie de* rouge.
   $\mathsf{G}((\neg r \wedge o \wedge \neg v) \rightarrow \mathsf{X}(r \wedge \neg o \wedge \neg v))$
3. *Le système passe infiniment souvent par la configuration* vert.
   $\mathsf{G}\, \mathsf{F}(\neg r \wedge \neg o \wedge v)$

Forme Normale Positive

Définition 16. *Une formule LTL qui n'utilise ni $\leftrightarrow$, ni $\rightarrow$, et dans laquelle toutes les négations ne portent que sur des propositions atomiques est dite* sous forme normale positive.

Toute formule LTL peut être réécrite en une formule LTL équivalente sous forme normale positive en utilisant les identités suivantes.

$$\begin{aligned}
\neg\top &= \bot & \neg\bot &= \top \\
\neg\neg f_1 &= f_1 & & \\
\neg(f_1 \vee f_2) &= \neg f_1 \wedge \neg f_2 & \neg(f_1 \wedge f_2) &= \neg f_1 \vee \neg f_2 \\
f_1 \rightarrow f_2 &= \neg f_1 \vee f_2 & f_1 \leftrightarrow f_2 &= (f_1 \wedge f_2) \vee (\neg f_1 \wedge \neg f_2) \\
\neg \mathsf{X}\, f_1 &= \mathsf{X}\neg f_1 & & \\
\neg(f_1 \,\mathsf{U}\, f_2) &= (\neg f_1)\,\mathsf{R}(\neg f_2) & \neg(f_1 \,\mathsf{R}\, f_2) &= (\neg f_1)\,\mathsf{U}(\neg f_2) \\
\neg \mathsf{G}\, f_1 &= \mathsf{F}\neg f_1 & \neg \mathsf{F}\, f_1 &= \mathsf{G}\neg f_1
\end{aligned}$$

Ces 5 dernières identités peuvent être facilement démontrées à partir des définitions des opérateurs temporels.

Exemple 8. *Voici des formules normales positives pour les trois propriétés de l'exemple 7 page précédente.*

1. $\mathsf{F}(\neg r \vee o \vee v)$
2. $\mathsf{G}(\neg r \vee o \vee v \vee \mathsf{X}(r \wedge \neg o \wedge \neg v))$
3. $\mathsf{G}\,\mathsf{F}(\neg r \wedge \neg o \wedge v)$

Bien que l'appellation *forme normale positive* soit la plus fréquemment utilisée, certains auteurs [58, 46] la désignent parfois sous le nom de *forme normale négative*.

Définitions récursives des opérateurs F, G, U et R. Les égalités suivantes peuvent être démontrées à partir des définitions des opérateurs temporels. Elles nous seront utiles lors de la traduction de formule LTL en automate de Büchi, chapitre 4.

$$\begin{aligned}
\mathsf{F}\, g &= g \vee \mathsf{X}\,\mathsf{F}\, g \\
\mathsf{G}\, g &= g \wedge \mathsf{X}\,\mathsf{G}\, g \\
f \,\mathsf{U}\, g &= g \vee (f \wedge \mathsf{X}(f \,\mathsf{U}\, g)) \\
f \,\mathsf{R}\, g &= g \wedge (f \vee \mathsf{X}(f \,\mathsf{R}\, g))
\end{aligned} \tag{3.1}$$

Ces réécritures, ainsi que les réécritures classiques de la logique propositionnelle, permettent d'exprimer toute formule LTL φ sous la forme

$$\varphi = \bigvee_i (P_i \wedge \mathsf{X}\, Q_i)$$

où les P_i sont des formules propositionnelles (c'est-à-dire sans opérateur temporel) et les Q_i sont des formules LTL. Lorsqu'on cherche à vérifier si une séquence σ vérifie une formule LTL, cette réécriture d'une formule LTL permet de séparer ce qui doit être vérifié à l'instant 0 (l'un des P_i) de ce qui doit être vérifié par la suite (le Q_i correspondant). Autrement dit:

$$\sigma \models \varphi \iff \exists i,\, \sigma(0) \models P_i \wedge \sigma^1 \models Q_i$$

Puisque nous travaillons sur des séquences infinies, signalons qu'il est incorrect d'appliquer la réécriture $f \mathsf{U} g = g \vee (f \wedge \mathsf{X}(f \mathsf{U} g))$ récursivement à l'infini car la formule produite accepterait une séquence vérifiant f à tous les instants sans jamais vérifier g. Or la sémantique de l'opérateur U implique que la formule g doit être vérifiée au bout d'un temps fini.

U et F sont les seuls opérateurs qui posent ce problème. Par la suite nous dirons que $f \mathsf{U} g$ et $\mathsf{F} g$ *promettent* de satisfaire g. Les identités (3.1) peuvent être utilisées infiniment à conditions de s'assurer que les séquences qu'elles permettent de reconnaître *tiennent leurs promesses*.

3.2.3 S1S: Logique monadique du second ordre à un successeur

S1S complète F1S avec les quantificateurs du second ordre, permettant de manipuler des ensembles d'instants.

Les formules de S1S sont construites à partir de:
- 0 : un symbole désignant l'instant initial
- $t+1$: un opérateur retournant l'instant successeur immédiat
- $t \leq u$: une relation d'ordre total sur les instants
- $\exists t, \forall t$: des quantificateurs du premier ordre
- $\exists^2 X, \forall^2 X$: des quantificateurs du second ordre
- $t \in X$: un test d'appartenance d'une variable du premier ordre à une variable du second
- AP: un ensemble de prédicats ($r(t)$, $o(t)$, $v(t)$, indiquant dans notre exemple si les feux sont allumés à l'instant t)
- $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$: les opérateurs usuels de la logique propositionnelle

La satisfaction d'une formule F1S φ par rapport à une séquence σ s'étend de façon évidente au second ordre, et nous noterons toujours $\mathscr{L}_{AP}(\varphi)$ l'ensemble des modèles d'une formule.

Exemple 9. *Avec ces définitions, nous sommes par exemple capables d'exprimer l'ensemble X de tous les instants pairs en disant que 0 appartient à l'ensemble, puis que, pour tout instant t de l'ensemble, on y trouve aussi $t+1+1$, mais pas $t+1$.*

$$\exists^2 X. \underbrace{(0 \in X \wedge (\forall t.(t \in X \rightarrow (\neg(t+1 \in X) \wedge (t+1+1 \in X)))))}_{\mathrm{Pair}(X)}$$

Si l'on note $\mathrm{Pair}(X)$ *cette contrainte sur l'ensemble X, la propriété «* le feu orange ne peut être allumé qu'aux instants impairs *» peut alors s'écrire:*

$$\exists^2 X.\mathrm{Pair}(X) \wedge \forall t.(t \in X \rightarrow \neg o(t))$$

Les propriétés basées sur la parité sont des exemples types de propriétés qui ne peuvent pas s'exprimer en F1S, et qui ne peuvent donc pas non plus s'exprimer en LTL.

Nous verrons section 3.3.2 que les automates de Büchi sont aussi expressifs que F1S, et permettent donc de représenter plus de propriétés que ne le peut LTL.

3.3 ω-Automates

En 1960, Büchi [23] présenta une procédure de décision pour S1S (3.2.3 page précédente) s'appuyant sur des automates acceptant des mots de longueur infinie.

26 ans plus tard Vardi [135] introduisait *l'approche automate pour la vérification LTL*, qui s'appuie sur les automates de Büchi pour vérifier des formules LTL sur un système.

Cette section introduit les différents ω-automates (c'est-à-dire automates acceptant des ω-mots) que nous manipulerons. Même s'ils acceptent des mots infinis, tous ces automates ont un nombre fini d'états. Nous commencerons par définir les structures de Kripke (section 3.3.1), dans lesquelles tous les chemins infinis correspondent à des mots acceptés. Ces structures sont utilisées pour représenter les systèmes à vérifier. Les automates de Büchi (section 3.3.2) imposent une condition supplémentaires sur les chemins infinis qui peuvent être acceptés. Nous généraliserons ensuite cette condition d'acceptation (section 3.3.3) avant de présenter enfin les automates que nous utiliserons en pratique (section 3.3.4).

3.3.1 Structure de Kripke

Une structure de Kripke est utilisée pour représenter les liens entre les états du système à vérifier. Il s'agit d'un graphe orienté dans lequel les nœuds, appelés états, sont étiquetés par les valuations de propriétés atomiques observées sur le système.

Définition 17. *Une* structure de Kripke *est un quintuplet* $K = \langle AP, \mathcal{Q}, q^0, \delta, l \rangle$ *où*
- AP *est une ensemble fini de propositions atomiques,*
- $\mathcal{Q}$ *est un ensemble fini d'états (les nœuds du graphe),*
- $q^0 \in \mathcal{Q}$ *est l'état initial,*
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ *est une fonction indiquant les états successeurs d'un état,*
- $l : \mathcal{Q} \mapsto 2^{AP}$ *est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état.*

La figure 2.4 page 21 donne un exemple de structure de Kripke avec $AP = \{r_1, r_2, d_1, d_2\}$ et la valeur de l indiquée sur chaque état.

Définition 18. *Un* chemin *(fini ou infini) dans une structure de Kripke* $K = \langle AP, \mathcal{Q}, q^0, \delta, l \rangle$ *est une séquence (finie ou infinie) d'états* $q_0 \cdot q_1 \cdot q_2 \cdots$ *telle que* $q_0 = q^0$ *et* $\forall i \geq 0$, $q_{i+1} \in \delta(q_i)$.

On notera $\mathrm{Run}(K)$ *l'ensemble des chemins infinis de* K.

Un chemin infini correspond à une exécution du système passant successivement par des configurations vérifiant les propositions $l(q_0), l(q_1), \ldots$. Une telle séquence $l(q_0) \cdot l(q_1) \cdots$ sera donc appelée une *exécution*.

Définition 19. *Le langage d'une structure de Kripke* K*, noté* $\mathscr{L}(K)$*, est l'ensemble des exécutions de* K.

$$\mathscr{L}(K) = \{l(q_0) \cdot l(q_1) \cdot l(q_2) \cdots \mid q_0 \cdot q_1 \cdot q_2 \cdots \in \mathrm{Run}(K)\}$$

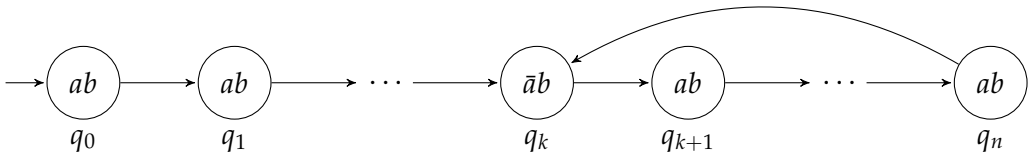


FIG. 3.2: Une structure de Kripke séquentielle $\langle \{a,b\}, \{q_0, q_1, \ldots, q_n\}, q_0, \delta, l \rangle$ avec $\forall i < n$, $\delta(q_i) = \{q_{i+1}\}$; $\delta(q_n) = \{q_k\}$; $\forall i \neq k, l(q_i) = ab$; et $l(q_k) = \bar{a}b$.

Définition 20. *Nous dirons qu'une structure de Kripke K vérifie une formule LTL φ, ce que l'on note $K \models \varphi$, si toutes ses exécutions vérifient K.*

$$K \models \varphi \iff \forall \sigma \in \mathscr{L}(K), \sigma \models \varphi$$

Nous chercherons parfois à représenter un chemin par une structure de Kripke. Comme le chemin infini traverse un nombre fini d'états, il existe nécessairement un point à partir duquel la séquence boucle. On appellera *structure de Kripke séquentielle* une telle structure.

Définition 21. *Une structure de Kripke $K = \langle AP, \mathcal{Q}, q^0, \delta, q \rangle$ est dite* séquentielle *si $\forall q \in \mathcal{Q}$, $|\delta(q)| = 1$.*

La figure 3.2 représente un exemple d'une telle structure.

3.3.2 Automates de Büchi

Définition 22. *Un automate de Büchi est un sextuplet $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta, l \rangle$ où*
- *Σ est un alphabet,*
- *$\mathcal{Q}$ est un ensemble fini d'états,*
- *$\mathcal{Q}^0 \subseteq \mathcal{Q}$ est un ensemble d'états initiaux,*
- *$\mathcal{F} \subseteq \mathcal{Q}$ est un ensemble d'états d'acceptation,*
- *$\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ est une fonction indiquant les états successeurs d'un état,*
- *$l : \mathcal{Q} \mapsto 2^{\Sigma} \setminus \{\emptyset\}$ est une fonction étiquetant chaque état par un ensemble non-vide de lettres de Σ.*

Définition 23. *Comme pour les structures de Kripke, on notera* $\mathrm{Run}(A)$ *l'ensemble des chemins infinis parcourant A à partir d'un état de $\mathcal{Q}^0$ tout en respectant δ:*

$$\mathrm{Run}(A) = \{q_0 \cdot q_1 \cdot q_2 \cdots \in \mathcal{Q}^\omega \mid q_0 \in \mathcal{Q}^0 \text{ et } \forall i \geq 0,\ q_{i+1} \in \delta(q_i)\}$$

Un chemin infini de A est acceptant s'il traverse des états d'acceptation infiniment souvent. Nous noterons $\mathrm{Acc}(A)$ *l'ensemble des chemins acceptants de A.*

$$\mathrm{Acc}(A) = \{r \in \mathrm{Run}(A) \mid \forall i \geq 0,\ \exists j \geq i,\ r(j) \in \mathcal{F}\}$$

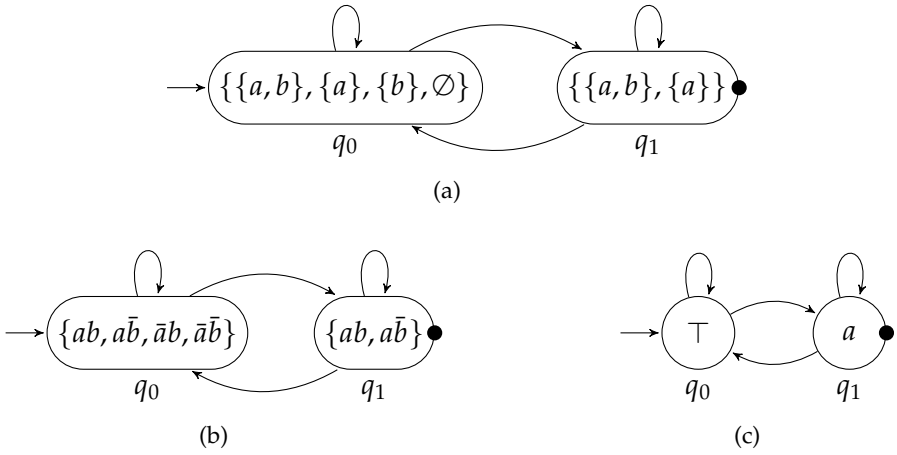


FIG. 3.3: Trois représentations équivalentes de l'automate de Büchi de l'exemple 10, en utilisant les équivalences de notations introduites dans la section 3.1.1 page 31. Nous n'utiliserons que la dernière, plus compacte, par la suite.

Définition 24. *Une séquence $\sigma \in \Sigma^\omega$ est une exécution de A si et seulement si il existe un chemin acceptant $q_0 \cdot q_1 \cdots \in \mathrm{Acc}(A)$ dont les étiquettes en contiennent les lettres: $\forall i \in \mathbb{N}, \sigma(i) \in l(q_i)$.*

Le langage d'un automate de Büchi A est l'ensemble de telles exécutions:

$$\mathscr{L}(A) = \{\sigma \in \Sigma^\omega \mid \exists q_0 \cdot q_1 \cdot q_2 \cdots \in \mathrm{Acc}(A), \forall i \in \mathbb{N}, \sigma(i) \in l(q_i)\}$$

Notons que la structure de Kripke $K = \langle AP, \mathcal{Q}, q^0, \delta, l\rangle$ possède le même langage que l'automate de Büchi $\langle 2^{AP}, \mathcal{Q}, \{q^0\}, \mathcal{Q}, \delta, l'\rangle$ dont tous les états sont acceptants et où $l'(x) = \{l(x)\}$.

Exemple 10. *Soit $AP = \{a, b\}$ deux propositions atomiques. L'ensemble des exécutions dans lesquelles la proposition a est vraie infiniment souvent (quelle que soit b), peut être représenté par l'automate de Büchi $\langle 2^{AP}, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \delta, l\rangle$ avec $\forall q$, $\delta(q) = \{q_0, q_1\}$, $l(q_0) = 2^{AP}$ et $l(q_1) = \{\{a, b\}, \{a\}\}$.*

Cet automate est représenté par la figure 3.3a, avec la convention que les états initiaux sont indiqués par « $\rightarrow$ » et les états acceptants sont marqués par un « ● ».

Il est cependant plus facile de raisonner avec les notations introduites dans la section 3.1.1 page 31, où 2^{AP} symbolise l'ensemble des configurations possibles des éléments de AP: $\{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$. La figure 3.3b donne cette interprétation. On y voit que pour n'importe quelle séquence de $(2^{AP})^\omega$ on peut trouver un chemin dans l'automate (par exemple en bouclant toujours sur l'état initial), mais qu'un tel chemin sera accepté seulement s'il passe infiniment souvent par q_1. Les seules séquences acceptées par cet automate sont donc celles qui contiennent une infinité de ab ou de $a\bar{b}$.

Des ensembles de configurations possibles, tels que $\{ab, a\bar{b}\}$, peuvent être interprétés comme des disjonctions de conjonctions. Par exemple $\{ab, a\bar{b}\}$ correspond à $(a \wedge b) \vee (a \wedge \neg b)$ qui se simplifie en a. Une troisième représentation de ces automates est donc d'étiqueter les états par des

formules de logique propositionnelle, comme dans la figure 3.3c. C'est cette dernière représentation, plus compacte, que nous adopterons par la suite.

Büchi [23] a montré que ces automates étaient aussi expressifs que S1S, la logique monadique du second ordre à un successeur (section 3.2.3 page 41). C'est-à-dire que pour toute formule S1S φ, il existe un automate de Büchi A_φ tel que $\mathscr{L}_{AP}(\varphi) = \mathscr{L}(A_\varphi)$ (et vice versa).

Exemple 11. *Le langage de l'automate de l'exemple 10 page précédente est* $\mathscr{L}_{AP}(\mathsf{G}\,\mathsf{F}\,a)$.

3.3.3 Automates de Büchi généralisés

Définition 25. *Un automate de Büchi généralisé (GBA) est un sextuplet* $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta, l \rangle$ *où*
- Σ *est un alphabet,*
- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $\mathcal{Q}^0 \subseteq \mathcal{Q}$ *est un ensemble d'états initiaux,*
- $\mathcal{F} \subseteq 2^{\mathcal{Q}}$ *est* un ensemble d'ensembles *d'états d'acceptation,*
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ *est une fonction indiquant les états successeurs d'un état,*
- $l : \mathcal{Q} \mapsto 2^{\Sigma} \setminus \{\emptyset\}$ *est une fonction étiquetant chaque état par un ensemble de lettres non-vide de* Σ.

La seule différence avec un automate de Büchi classique est que l'ensemble $\mathcal{F}$ est maintenant un ensemble d'ensembles d'états. Un chemin ne sera acceptant que s'il passe infiniment souvent par un état de chacun de ces ensembles.

Définition 26. *L'ensemble des chemins acceptants d'un automate de Büchi généralisé* $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta, l \rangle$ *est*

$$\mathrm{Acc}(A) = \{r \in \mathrm{Run}(A) \mid \forall F \in \mathcal{F},\, \forall i \geq 0,\, \exists j \geq i,\, r(j) \in F\}$$

et le langage associé à l'automate est toujours l'ensemble des exécutions pour lesquelles il existe un chemin acceptant:

$$\mathscr{L}(A) = \{\sigma \in \Sigma^\omega \mid \exists q_0 \cdot q_1 \cdot q_2 \cdots \in \mathrm{Acc}(A),\, \forall i \in \mathbb{N},\, \sigma(i) \in l(q_i)\}$$

Naturellement un automate de Büchi (non généralisé) $\langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta, l \rangle$ peut être vu comme un automate de Büchi généralisé $\langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \{\mathcal{F}\}, \delta, l \rangle$ possédant un seul ensemble d'états acceptants.

Exemple 12. *L'automate de Büchi* $\langle 2^{\{a,b\}}, \{q_0, q_1, q_2\}, \{q_0, q_1, q_2\}, \{\{q_1\}, \{q_2\}\}, \delta, l \rangle$ *avec* $\delta : x \mapsto \{q_0, q_1, q_2\}$, $l(q_0) = \top$, $l(q_1) = a$, *et* $l(q_2) = b$ *accepte le même langage que la formule LTL* $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$ *(infiniment souvent a et infiniment souvent b).*

Les deux conditions d'acceptation représentées sur la figure 3.4 page suivante par « $\bullet$ *» et «* $\circ$ *», assurent que tout chemin accepté passe infiniment souvent aussi bien par* q_1 *que par* q_2.

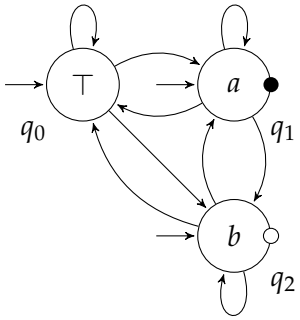


FIG. 3.4: Automate de Büchi généralisé de l'exemple 12 page précédente, acceptant toutes les exécutions dans lesquelles a et b sont vrais infiniment souvent. Il correspond à la formule LTL $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$.

Dégénéralisation. Un automate de Büchi généralisé $A = \langle \Sigma, Q, Q^0, \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{r-1}\}, \delta, l\rangle$ peut être converti en un automate de Büchi non-généralisé (définition 22 page 43) $A' = \langle \Sigma, Q', Q'^0, \mathcal{F}', \delta', l'\rangle$ par la construction suivante appelée *dégénéralisation* [30, section 9.2.2]:

- $Q' = Q \times [\![0, r]\!]$
- $Q'^0 = Q \times \{0\}$
- $\mathcal{F}' = Q \times \{r\}$
- $\forall (q, j) \in Q',\ l'((q, j)) = l(q)$
- $\forall (q, j) \in Q',\ \delta'((q, j)) = \{(q', acc_j(q)) \mid q' \in \delta(q)\}$ avec $acc_j(q) = \begin{cases} 0 & \text{si } j = r \\ j+1 & \text{si } q \in \mathcal{F}_j \\ j & \text{sinon} \end{cases}$

Si l'automate A possède n états accessibles, A' en possède donc au pire $n(r+1)$.

Exemple 13. *Cette construction est illustrée figure 3.5 page ci-contre. Pour construire l'automate dégénéralisé correspondant à l'automate généralisé de la figure 3.5a, les états de l'automate sont d'abord dupliqués $|\mathcal{F}| + 1$ fois (figure 3.5b) en ne gardant les états initiaux que sur la première copie. Dans la i^e copie, nous nous arrangeons ensuite pour sauter dans la copie suivante à partir de chaque état de l'ensemble d'acceptation $\mathcal{F}_i$ (figure 3.5c). Les états de la dernière copie sont tous connectés à la première et marqués comme états acceptants (figure 3.5d). Enfin, nous pouvons omettre les états non-accessibles (figure 3.5e).*

L'exemple 13 montre bien la logique de cette construction. Pour qu'un chemin soit accepté par l'automate dégénéralisé, il faut qu'il passe infiniment souvent par l'un des états acceptants qui se trouvent dans la dernière copie de l'automate. Pour arriver là, il faut nécessairement avoir franchi les copies intermédiaires, et cela n'est possible qu'à condition d'être passé par chacun des ensembles d'acceptation de l'automate d'origine.

Cette construction peut être améliorée à partir des deux constats suivants:
- Lorsqu'un état de A appartient à plusieurs ensembles d'acceptation, les transitions sortantes des états correspondants dans l'automate dégénéralisé peuvent franchir plusieurs niveaux d'un coup.

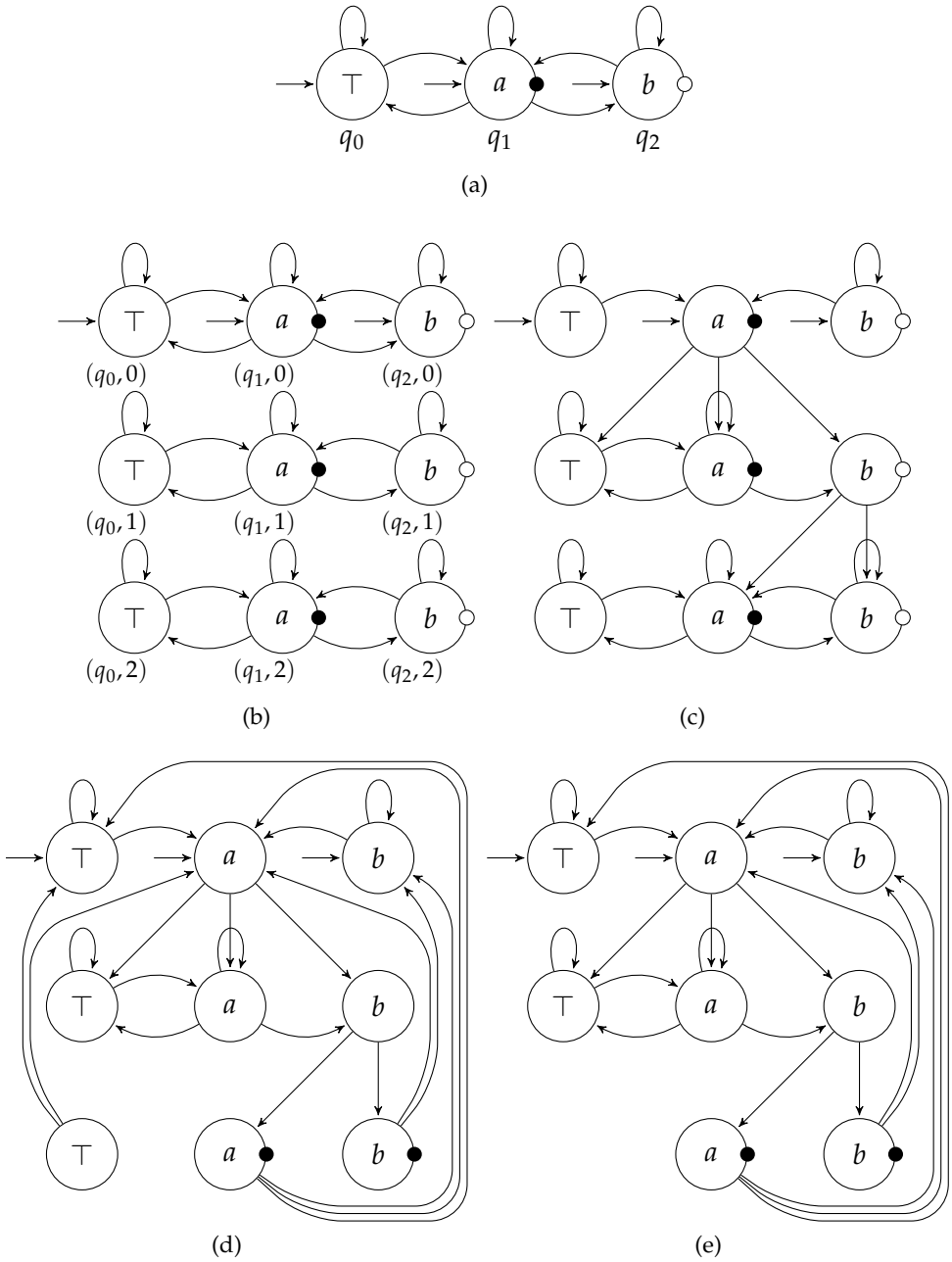


FIG. 3.5: Étapes de la dégénéralisation d'un automate de Büchi généralisé. Voir l'exemple 13 page précédente.

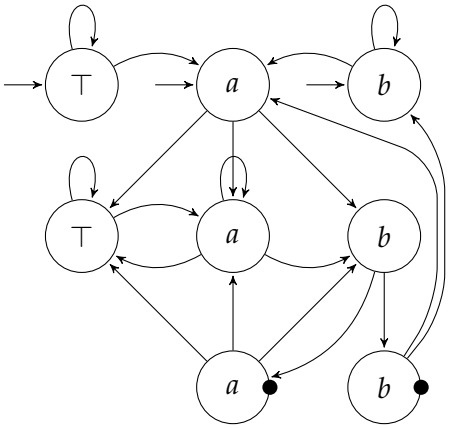


FIG. 3.6: Amélioration de la dégénéralisation de la figure 3.5e page précédente.

– La dernière copie de l'automate peut ne pas être connectée directement à la première, mais aux suivantes en fonction des conditions d'acceptation de chaque état. Ainsi l'état $(q_1, 2)$ de la figure 3.5e pourrait être connecté aux états $(q_0, 1), (q_1, 1), (q_2, 1)$ plutôt que $(q_0, 0), (q_1, 0), (q_2, 0)$. La figure 3.6 montre ce changement.

Ces remarques donnent une nouvelle définition pour acc_j:

$$acc_j(q) = \begin{cases} j & \text{si } j < r \text{ et } q \notin \mathcal{F}_j, \\ \max\{n \in [\![j, r]\!] \mid \forall k \in [\![j, n]\!], q \in \mathcal{F}_k\} & \text{si } j < r \text{ et } q \in \mathcal{F}_j, \\ 0 & \text{si } j = r \text{ et } q \notin \mathcal{F}_0, \\ \max\{n \in [\![0, r]\!] \mid \forall k \in [\![0, n]\!], q \in \mathcal{F}_k\} & \text{si } j = r \text{ et } q \in \mathcal{F}_0 \end{cases}$$

Cette dernière construction semble préférable car elle abrège les cycles au sein de l'automate, ce qui devrait favoriser les algorithmes d'*emptiness check* utilisé par la suite. Ces algorithmes que nous présenterons chapitre 5 cherchent des cycles dans l'automate. Oddoux [100], section 6.1.2 évoque un autre type de dégénéralisation dans laquelle, au lieu de créer r copies de l'automate et de traverser les conditions d'acceptation une à une, il crée 2^r copies représentant chacune un ensemble de conditions d'acceptation traversées. Cette méthode, parce qu'elle n'impose pas d'ordre sur le parcours des conditions d'acceptation, permet de créer des contre-exemples plus courts, mais cela se fait au détriment de la taille de l'automate: l'accroissement exponentiel de la taille de l'automate n'est pas raisonnable.

3.3.4 Conditions d'acceptation sur les transitions: TGBA

Définition 27. *Un* automate de Büchi généralisé étiqueté sur les transitions *(TGBA) est un automate de Büchi dans lequel les étiquettes sont portées par les transitions et où les conditions d'acceptation de Büchi généralisées portent sur les transitions. C'est-à-dire un quintuplet* $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta \rangle$ *où*

– Σ *est un alphabet,*

- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $\mathcal{Q}^0 \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
- $\mathcal{F}$ *est un ensemble fini d'éléments appelés conditions d'acceptation,*
- $\delta \subseteq \mathcal{Q} \times (2^\Sigma \setminus \{\emptyset\}) \times 2^\mathcal{F} \times \mathcal{Q}$ *est la relation de transition de l'automate (chaque transition étant étiquetée par une formule propositionnelle ainsi qu'un ensemble de conditions d'acceptation).*

L'acronyme TGBA, dont nous ferons une utilisation intensive, signifie *Transition-based Generalized Büchi Automaton*. Il a été introduit par Giannakopoulou et Lerda [63] en 2002, cependant de tels automates étaient utilisés bien avant: Couvreur [34] s'en est servi pour sa traduction de formules LTL, sur laquelle nous reviendrons section 4.2 page 86, et même Michel [96] introduisait en 1984 une structure similaire (cf. section 4.1.2 page 68).

À la différence des automates étiquetés sur les états, les chemins des TGBA sont des séquences de transitions, c'est-à-dire des éléments de δ^∞. Il nous sera utile de pouvoir désigner les différents champs d'une transitions t. Nous noterons donc t^{in}, t^{prop}, t^{acc}, t^{out}, l'état source, la formule propositionnelle, les conditions d'acceptation, et l'état destination d'une transition $t \in \delta$. Autrement dit $t = (t^{\text{in}}, t^{\text{prop}}, t^{\text{acc}}, t^{\text{out}})$.

Définition 28. *Comme pour les autres automates,* $\text{Run}(A)$ *désigne l'ensemble des chemins infinis parcourant un TGBA* $A = \langle \Sigma, \mathcal{Q}, q^0, \mathcal{F}, \delta \rangle$ *à partir d'un état initial tout en respectant sa relation de transition* δ*:*

$$\text{Run}(A) = \{t_0 \cdot t_1 \cdot t_2 \cdots \in \delta^\omega \mid t_0^{\text{in}} \in \mathcal{Q}^0,\ \forall i \geq 0,\ t_i^{\text{out}} = t_{i+1}^{\text{in}}\}$$

Les chemins acceptants sont ceux dont les transitions visitent infiniment chaque condition d'acceptation:

$$\text{Acc}(A) = \{r \in \text{Run}(A) \mid \forall f \in \mathcal{F},\ \forall i \geq 0,\ \exists j \geq i,\ f \in r(j)^{\text{acc}}\}$$

et le langage associé à l'automate reste l'ensemble des exécutions pour lesquelles il existe un chemin acceptant:

$$\mathscr{L}(A) = \{\sigma \in \Sigma^\omega \mid \exists t_0 \cdot t_1 \cdot t_2 \cdots \in \text{Acc}(A),\ \forall i \in \mathbb{N},\ \sigma(i) \in t_i^{\text{prop}}\}$$

Définition 29. *Pour un TGBA* $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta \rangle$*. Nous noterons* $\text{Reach}(\mathcal{A})$ *l'ensemble des états accessibles, c'est-à-dire:*

$$\text{Reach}(\mathcal{A}) = \{q \in \mathcal{Q} \mid \exists \sigma \in \text{Run}(A),\ \exists i \geq 0,\ \sigma(i)^{\text{in}} = q\}$$

Exemple 14. *Le TGBA* $\langle 2^{\{a,b\}}, \{q\}, \{q\}, \{f, g\}, \delta \rangle$ *avec*

$$\delta = \{(q, 2^{\{a,b\}}, \emptyset, q), (q, \{\{a,b\}, \{a\}\}, f, q), (q, \{\{a,b\}, \{b\}\}, g, q)\}$$

accepte le même langage que la formule LTL $\mathsf{GF}a \wedge \mathsf{GF}b$ *(infiniment souvent* a *et infiniment souvent* b*).*

Les deux conditions d'acceptation f *et* g*, représentées sur la figure 3.7 page suivante par «* $\bullet$ *» et «* $\circ$ *», assurent que tout chemin accepté passe infiniment souvent par l'une et l'autre de ces transitions.*

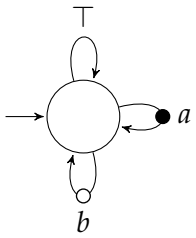


FIG. 3.7: TGBA de l'exemple 14 page précédente, acceptant toutes les exécutions dans lesquelles a et b sont vrais infiniment souvent. Il correspond à la formule LTL $\mathsf{GF}\,a \wedge \mathsf{GF}\,b$. À comparer à la version « sur états » de la figure 3.4 page 46.

Par le passé [44, 37], nous n'avons utilisé que des TGBA avec un seul état initial. L'implémentation des TGBA dans la bibliothèque Spot (annexe A page 201) ne supporte qu'un état initial et cela n'a jamais été un frein. Supporter un ensemble d'état initiaux n'est pas indispensable en pratique: les TGBA que nous créerons à partir de formules LTL n'ont qu'un état initial, et les structures de Kripke que nous interpréterons comme des TGBA sans condition d'acceptation n'en demandent pas davantage. D'autre part il est toujours possible de réécrire un automate ayant plusieurs états initiaux en un automate possédant un unique état: il suffit de créer ce nouvel état initial et de le connecter au reste de l'automate avec des copies des transitions sortantes des anciens états initiaux. Si nous avons décidé de travailler avec un ensemble d'états initiaux, c'est principalement à cause de la proposition 19 page 147, mais aussi pour homogénéiser GBA et TGBA (facilitant par exemple l'écriture de la proposition 3 page suivante).

Définition 30. *Soit $A = \langle \Sigma, Q, Q^0, \mathcal{F}, \delta \rangle$ un TGBA, et $\mathcal{T} \subseteq Q$ un état de A. Nous noterons $A[\mathcal{T}]$ l'automate partageant la même structure que A, mais utilisant $\mathcal{T}$ comme ensemble d'états initiaux.*

Autrement dit, $A[\mathcal{T}] = \langle \Sigma, Q, \mathcal{T}, \mathcal{F}, \delta \rangle$.

Proposition 1. *Soit $A = \langle \Sigma, Q, q^0, \mathcal{F}, \delta \rangle$ un TGBA tel que $\mathrm{Acc}(A) \neq \emptyset$ alors il existe $\sigma \in \mathrm{Acc}(A), i \geq 0$ et $k > 0$ tels que $\sigma = \sigma(0) \cdot \sigma(1) \cdots \sigma(i-1) \cdot (\sigma(i) \cdot \sigma(i+1) \cdots \sigma(i+k-1))^\omega$.*

Autrement dit tout automate acceptant possède forcément un chemin acceptant cyclique, c'est-à-dire de la forme $\sigma(0) \cdot \sigma(1) \cdots \sigma(i-1) \cdot (\sigma(i) \cdot \sigma(i+1) \cdots \sigma(i+k-1))^\omega$. Ainsi l'automate de la figure 3.7 reconnaît le mot $(a\bar{b} \cdot \bar{a}b)^\omega$ avec un chemin acceptant cyclique évident. Cependant tout chemin acceptant n'est pas nécessairement cyclique. Par exemple sur ce même automate, aucun des chemins acceptants qui reconnaissent le mot suivant ne sont cycliques au sens de la propriété 1:

$$a\bar{b} \cdot \bar{a}b \cdot \underbrace{a\bar{b} \cdot a\bar{b}}_{2 \text{ fois}} \cdot \underbrace{\bar{a}b \cdot \bar{a}b}_{2 \text{ fois}} \cdots \underbrace{a\bar{b} \cdot a\bar{b} \cdots a\bar{b}}_{i \text{ fois}} \cdot \underbrace{\bar{a}b \cdot \bar{a}b \cdots \bar{a}b}_{i \text{ fois}} \cdots$$

Démonstration. La propriété découle du fait qu'un chemin acceptant $\sigma' \in \mathrm{Acc}(A) \subseteq \delta^\omega$ est un chemin infini qui ne peut utiliser qu'un nombre fini $|\delta|$ de transitions différentes. Pour chaque condition d'acceptation f, il existe forcément une transition étiquetée par

f qui sera visitée infiniment souvent. Chacune de ces transitions étant visitée infiniment souvent, elles sont forcément accessibles les unes des autres et l'on peut construire un circuit qui les visite toutes. Ce circuit étant accessible depuis l'état initial, il est donc bien possible de construire un chemin acceptant σ sous la forme indiquée à partir des transitions de σ'. □

Proposition 2. *Ajouter ou retirer une condition d'acceptation à une transition qui ne peut intervenir dans aucun circuit d'un TGBA ne change pas le langage accepté par cet automate.*

Démonstration. Si une transition n'intervient dans aucun cycle, elle ne peut-être répétée infiniment souvent et ne peut donc pas changer l'acceptation des chemins qui l'empruntent. Comme le langage est défini à partir de l'ensemble des exécutions acceptées, celui-ci reste inchangé. □

3.3.5 Lien entre TGBA et GBA

Un GBA peut être converti en TGBA en « poussant » les étiquettes de chaque état vers les transitions sortantes de ces états. Les conditions d'acceptation sont ici à interpréter aussi comme des étiquettes: pour chaque ensemble d'acceptation F du GBA nous étiquetterons une transition du TGBA par f si son état source appartient à f dans le GBA.

Proposition 3. *Soit $A = \langle \Sigma, Q, Q^0, \mathcal{F}, \delta, l \rangle$ un GBA (déf. 25 page 45). Le TGBA (déf. 27 page 48) $B = \langle \Sigma, Q, Q^0, \mathcal{F}, \delta' \rangle$ où*

$$\delta' = \big\{ (t^{\text{in}}, t^{\text{prop}}, t^{\text{acc}}, t^{\text{out}}) \in Q \times 2^{\Sigma} \times 2^{\mathcal{F}} \times Q \mid t^{\text{out}} \in \delta(t^{\text{in}}), \; t^{\text{prop}} = l(t^{\text{in}}), \text{ et } t^{\text{acc}} = \{ f \in \mathcal{F} \mid t^{\text{in}} \in f \} \big\}.$$

est tel que $\mathscr{L}(A) = \mathscr{L}(B)$.

La figure 3.8 page suivante montre le TGBA créé de cette façon à partir du GBA de l'exemple 12 page 45. La taille de l'automate, autant en terme d'états que de transitions, est identique. Dans la figure figure 3.7 page précédente nous avions cependant déjà présenté un TGBA plus concis pour représenter le même langage avec un seul état.

Proposition 4. *Il n'existe pas de GBA avec un seul état acceptant le langage de la formule LTL* $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$.

Démonstration. Supposons qu'il existe un GBA $A = \langle \{a,b\}, \{q\}, \{q\}, \mathcal{F}, \delta, l \rangle$ acceptant $\mathscr{L}_{\{a,b\}}(\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b)$.

Nous avons forcément $l(q) = \mathcal{F}$, sinon il ne serait pas possible de construire un chemin acceptant.

Comme $(a\bar{b} \cdot \bar{a}b)^{\omega} \in \mathscr{L}_{\{a,b\}}(\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b)$, il existe une transition $(q, P, q) \in \delta$ telle que $\{a\} \in P$, c'est-à-dire une transition reconnaissant $a\bar{b}$.

Le chemin de l'automate construit en franchissant cette transition infiniment souvent est acceptant. Cependant ce chemin accepte l'exécution $(a\bar{b})^{\omega} \notin \mathscr{L}_{\{a,b\}}(\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b)$, ce qui contredit l'hypothèse. A n'existe donc pas. □

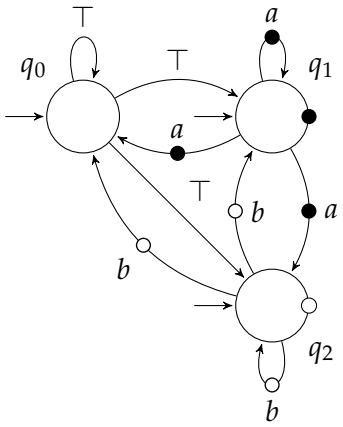


FIG. 3.8: TGBA construit à partir de l'automate de Büchi généralisé de la figure 3.4 page 46, en poussant les étiquettes comme indiqué dans la proposition 3 page précédente. Ce TGBA est moins compact que celui de la figure 3.7 page 50, mais reconnaît le même langage.

La proposition suivante donne une façon de construire un GBA à partir d'un TGBA en poussant toutes les étiquettes d'une transition sur l'état de destination. Comme les transitions arrivant sur un même état peuvent porter des propositions ou conditions d'acceptation différentes, il est nécessaire de dupliquer chaque état.

Proposition 5. *Soit* $B = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta \rangle$ *un TGBA.*

Le GBA $A = \langle \Sigma, \mathcal{Q}', \mathcal{Q}'^0, \mathcal{F}', \delta', l \rangle$ *où*
– $\mathcal{Q}' = \mathcal{Q} \times 2^\Sigma \times 2^\mathcal{F}$
– $\forall (q', P, F) \in \mathcal{Q}', \big((q', P, F) \in \mathcal{Q}'^0 \iff \exists q \in \mathcal{Q}^0, (q, P, F, q') \in \delta\big)$
– $\forall ((q, P, F), (q', P', F')) \in \mathcal{Q}'^2, \big(((q, P, F), (q', P', F')) \in \delta' \iff (q, P', F', q') \in \delta\big)$
– $\mathcal{F}' = \big\{ \{(q, P, F) \in \mathcal{Q}' \mid F = G\} \,\big|\, G \in \mathcal{F} \big\}$
– $l\big((q, P, F)\big) = P$
est tel que $\mathscr{L}(A) = \mathscr{L}(B)$.

Cette construction n'est clairement pas optimale: par exemple le TGBA de la figure 3.8 devient un GBA à 9 états, au lieu des trois de la figure 3.4 page 46.

Les propositions 3 et 5 permettent d'affirmer que les TGBA sont aussi expressifs que les GBA. Des propositions 3 et 4 nous pouvons conclure que les TGBA sont plus concis (en nombre d'états) que les GBA.

Il existe un type d'automate intermédiaire entre le GBA et le TGBA en terme de concision: il s'agit d'automates dans lesquels les transitions sont étiquetées par des lettres de Σ, mais où les conditions d'acceptation portent toujours sur les états. Ces automates peuvent aussi être convertis en GBA ou TGBA en poussant les étiquettes qui ne sont pas à leur place sur les transitions ou l'état suivant, selon le cas.

3.3.6 Opérations sur les TGBA

On sait, grâce à Büchi [23], que la classe des langages acceptés par les automates de Büchi (au sens de la définition 22 page 43) est close par intersection, union et complémentation. Comme il est possible de transformer un tel automate en TGBA et vice-versa (propositions 3 et 5), ces résultats s'étendent bien sûr aux TGBA. Dans cette section nous définissons ces opérations directement sur des TGBA.

Produit et somme. Les opérations d'intersection et d'union de langages se définissent facilement lorsque les deux automates partagent le même alphabet. Nous nous restreignons à ce cas dans un premier temps.

Définition 31. *Soient* $A_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{Q}_1^0, \mathcal{F}_1, \delta_1 \rangle$ *et* $A_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{Q}_2^0, \mathcal{F}_2, \delta_2 \rangle$ *deux TGBA partageant le même ensemble de propositions atomiques.*

Le produit synchronisé *de* A_1 *et* A_2 *est l'automate noté* $A_1 \otimes A_2 = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta_1 \rangle$ *où*
- $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$
- $\mathcal{Q}^0 = \mathcal{Q}_1^0 \times \mathcal{Q}_2^0$
- $\mathcal{F} = (\mathcal{F}_1 \times \{1\}) \cup (\mathcal{F}_2 \times \{2\})$
- $\delta = \{((t_1^{\text{in}}, t_2^{\text{in}}), t_1^{\text{prop}} \cap t_2^{\text{prop}}, (t_1^{\text{acc}} \times \{1\}) \cup (t_2^{\text{acc}} \times \{2\}), (t_1^{\text{out}}, t_2^{\text{out}})) \mid$
$t_1 \in \delta_1,\ t_2 \in \delta_2,\ t_1^{\text{prop}} \cap t_2^{\text{prop}} \neq \emptyset\}$

Le seul intérêt des produits dans la définition de $\mathcal{F}$ ci-dessus est d'éviter d'imposer $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$. Dans la pratique les conditions d'acceptation sont souvent des entiers et, si les automates proviennent de sources indépendantes, il peut y avoir un conflit. On renomme donc ces conditions d'acceptation de façon à en garantir l'unicité.

Proposition 6. *Soient* A_1 *et* A_2 *les deux TGBA de la définition 31, alors* $\mathscr{L}(A_1 \otimes A_2) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$.

Définition 32. *Soient* $A_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{Q}_1^0, \mathcal{F}_1, \delta_1 \rangle$ *et* $A_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{Q}_2^0, \mathcal{F}_2, \delta_2 \rangle$ *deux TGBA partageant le même ensemble de propositions atomiques.*

La somme de A_1 *et* A_2 *est l'automate noté* $A_1 \oplus A_2$ *et défini par* $A_1 \oplus A_2 = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta_1 \rangle$ *avec*
- $\mathcal{Q} = (\mathcal{Q}_1 \times \{1\}) \cup (\mathcal{Q}_2 \times \{2\})$
- $\mathcal{Q}^0 = (\mathcal{Q}_1^0 \times \{1\}) \cup (\mathcal{Q}_2^0 \times \{2\})$
- $\mathcal{F} = (\mathcal{F}_1 \times \{1\}) \cup (\mathcal{F}_2 \times \{1\})$
- *et* δ *est défini par*

$$\begin{aligned}\delta = \{&((t_1^{\text{in}}, 1), t_1^{\text{prop}}, (t_1^{\text{acc}} \times \{1\}) \cup (\mathcal{F}_2 \times \{2\}), (t_1^{\text{out}}, 1)) \mid t_1 \in \delta_1\} \\ \cup \{&((t_2^{\text{in}}, 2), t_2^{\text{prop}}, (\mathcal{F}_1 \times \{1\}) \cup (t_2^{\text{acc}} \times \{2\}), (t_2^{\text{out}}, 2)) \mid t_2 \in \delta_2\}\end{aligned}$$

Proposition 7. *Soient* A_1 *et* A_2 *les deux TGBA de la définition 32, alors* $\mathscr{L}(A_1 \oplus A_2) = \mathscr{L}(A_1) \cup \mathscr{L}(A_2)$.

Intéressons nous maintenant au cas où les deux automates ne partagent pas exactement le même alphabet. Si cet alphabet est basé sur des valuations de propositions atomiques, il est toujours possible de l'étendre *a posteriori*.

Par exemple considérons le TGBA

$$A_1 = \langle 2^{\{a,b\}}, \{q_0, q_1\}, \{q_0\}, \{f\}, \{(q_0, \{a\bar{b}\}, \emptyset, q_1), (q_0, \{ab, \bar{a}b\}, \{f\}, q_1), (q_1, 2^{\{a,b\}}, \{f\}, q_1)\}\rangle$$

représenté figure 3.9a page suivante, et l'automate

$$A_2 = \langle 2^{\{b,c\}}, \{q_0, q_1\}, \{q_0\}, \emptyset, \{(q_0, \{\bar{a}c, \bar{a}\bar{c}\}, \emptyset, q_0), (q_0, \{ac, a\bar{c}\}, \emptyset, q_1), (q_1, \{ac\}, \emptyset, q_1), (q_1, \{\bar{a}c\}, \emptyset, q_0)\}\rangle$$

représenté figure 3.9b. L'automate A_1 accepte le langage $L_1 = \mathscr{L}_{\{a,b\}}(a \mathsf{U} b)$ tandis que A_2 accepte $L_2 = \mathscr{L}_{\{a,c\}}(\mathsf{G}(a \rightarrow \mathsf{X}\, c))$.

On peut maintenant se demander quel est le langage $L \subseteq (2^{\{a,b,c\}})^\omega$ formé des ω-mots dont les projections (au sens de la définition 11 page 35) sur $2^{\{a,b\}}$ et $2^{\{a,c\}}$ sont acceptées par les automates correspondants. Il suffirait pour cela d'étendre l'alphabet des automates A_1 et A_2 afin qu'ils produisent tous les mots de $2^{\{a,b,c\}}$ tels que leur projection sur $2^{\{a,b\}}$ et $2^{\{b,c\}}$ reste dans le langage d'origine (L_1 et L_2).

Définition 33. *Soit $A = \langle 2^{AP}, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta\rangle$ un TGBA, et $AP' \supseteq AP$ un ensemble de propositions atomiques plus large.*

On notera $\text{extend}_{AP'}(A)$ *l'automate $\langle 2^{AP'}, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta'\rangle$ où*

$$\delta' = \{(t^{\text{in}}, t^{\text{prop}} \times (AP' \setminus AP), t^{\text{acc}}, t^{\text{out}}) \mid (t^{\text{in}}, t^{\text{prop}}, t^{\text{acc}}, t^{\text{out}}) \in \delta\}$$

Proposition 8. *Avec les définitions ci-dessus*

$$\mathscr{L}(A) = \emptyset \implies \mathscr{L}(A') = \emptyset$$

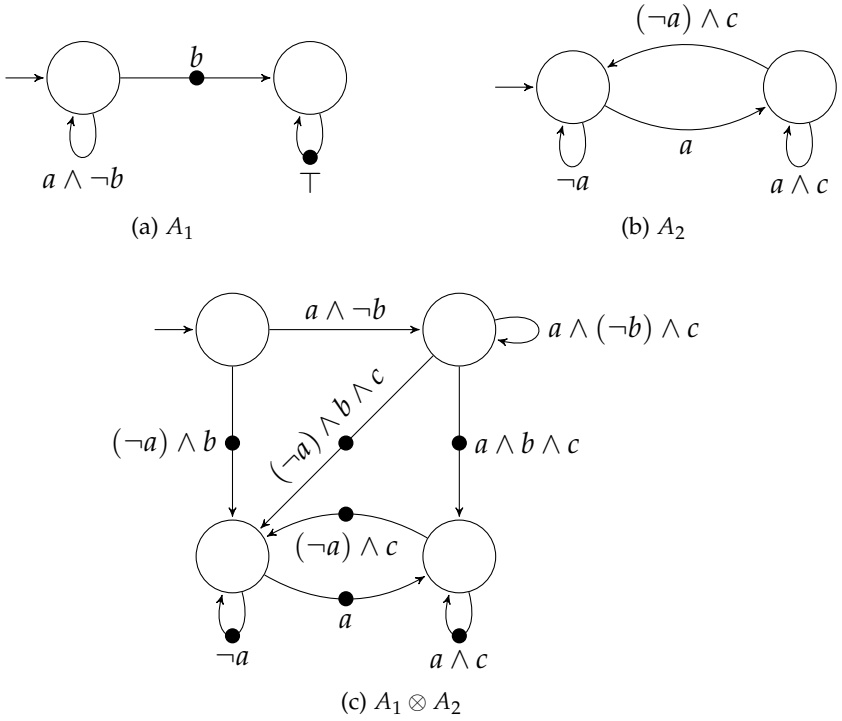
$$\mathscr{L}(A) \neq \emptyset \implies \begin{cases} \mathscr{L}(A')_{|AP} = \mathscr{L}(A) \\ \mathscr{L}(A')_{|AP' \setminus AP} = (2^{AP' \setminus AP})^\omega \end{cases}$$

Notons que les TGBA A et A' ont la même représentation lorsque nous utilisons des formules propositionnelles pour étiqueter de façon symbolique les arcs sur les figures. Par exemple $a \vee b$ représente aussi bien l'ensemble $\{ab, \bar{a}b, a\bar{b}\}$ pour $AP = \{a, b\}$ que l'ensemble $\{abc, \bar{a}bc, a\bar{b}c, ab\bar{c}, \bar{a}b\bar{c}, a\bar{b}\bar{c}\}$ pour $AP' = \{a, b, c\}$.

Définition 34. *Soient $A_1 = \langle 2_1^{AP}, \mathcal{Q}_1, q_1^0, \mathcal{F}_1, \delta_1\rangle$ et $A_2 = \langle 2_2^{AP}, \mathcal{Q}_2, q_2^0, \mathcal{F}_2, \delta_2\rangle$ deux TGBA. Posons $AP = AP_1 \cup AP_2$. Par extension, on notera*

$$A_1 \otimes A_2 = \text{extend}_{AP}(A_1) \otimes \text{extend}_{AP}(A_2)$$
$$A_1 \oplus A_2 = \text{extend}_{AP}(A_1) \oplus \text{extend}_{AP}(A_2)$$

Proposition 9. *Soient A_1 et A_2 les deux TGBA ci-dessus, alors*
– $\sigma \in \mathscr{L}(A_1 \otimes A_2) \iff \sigma_{|2^{AP_1}} \in \mathscr{L}(A_1) \wedge \sigma_{|2^{AP_2}} \in \mathscr{L}(A_2)$
– $\sigma \in \mathscr{L}(A_1 \oplus A_2) \iff \sigma_{|2^{AP_1}} \in \mathscr{L}(A_1) \vee \sigma_{|2^{AP_2}} \in \mathscr{L}(A_2)$



(a) A_1 (b) A_2

(c) $A_1 \otimes A_2$

FIG. 3.9: (a) TGBA acceptant le langage de la formule LTL $a \mathsf{U} b$. (b) TGBA acceptant $\mathsf{G}(a \rightarrow \mathsf{X} c)$. (c) produit des deux automates précédents, au sens de la définition 34 page ci-contre, ce produit accepte le langage de la formule $(a \mathsf{U} b) \wedge \mathsf{G}(a \rightarrow \mathsf{X} c)$.

Négation. La complémentation d'un automate de Büchi (et *a fortiori* d'un TGBA) est une opération complexe. En 1960, Büchi [23] a proposé une construction de complexité doublement exponentielle : par cette construction, la négation d'un automate de n états posséderait $2^{2^{O(n)}}$ états. Il faudra attendre Safra [104, 105] et Klarlund [85] pour voir les premiers algorithmes en $2^{O(n \log n)}$, la borne inférieure théorique. Les structures de données utilisées par cet algorithme sont assez compliquées et ses implémentations sont peu nombreuses. L'opération semble avoir suscité un regain d'intérêt récemment, avec plusieurs publications visant à la simplifier ou l'améliorer [85, 130, 66, 53]. Vardi [138, 140] donne un aperçu des progrès effectués dans ce domaine. Malgré tout, la complémentation reste une opération impraticable sur des automates de taille supérieure à quelques états: lors de leur comparaison de différentes implémentations, Tabakov et Vardi [114] ont rencontré des problèmes à partir de 6 états.

Dégénéralisation. La dégénéralisation d'un TGBA peut se faire de façon similaire à celle des automates de Büchi basés sur les états présentée section 3.3.3 page 46. Par contre, elle n'a besoin que de n copies de l'automate et non $n+1$. Comme dans la dégénéralisation sur les états, lorsqu'on est dans la i^{e} copie de l'automate, seul le franchissement d'une transition étiquetée par la i^{e} condition d'acceptation permet de passer à la copie suivante.

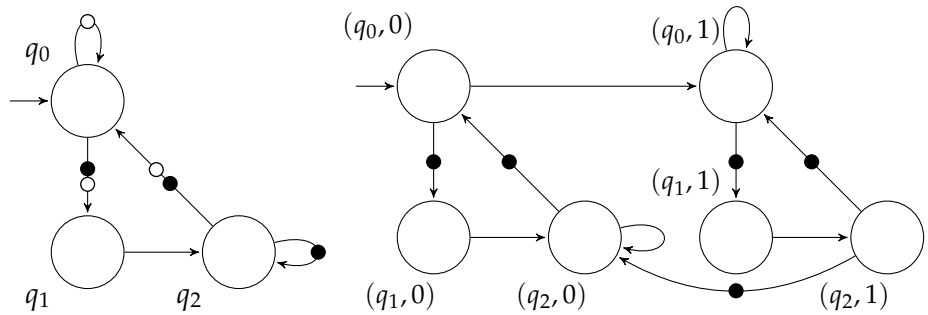


FIG. 3.10: Exemple de dégénéralisation d'un TGBA. Les formules propositionnelles qui étiquettent les transitions n'apparaissent pas sur les figures. (a) TGBA original avec pour conditions d'acceptation $\mathcal{F} = \{\circ, \bullet\}$, considérées dans cet ordre. (b) automate dégénéralisé obtenu par la construction de la proposition 10.

Si l'on passe infiniment souvent de la dernière copie à la première, c'est que l'on visite toutes les conditions d'acceptation de l'automate d'origine infiniment souvent.

Proposition 10. *Soit $A = \langle AP, Q, q^0, \{f_0, f_1, \ldots, f_{r-1}\}, \delta\rangle$ un TGBA quelconque. L'automate $A' = \langle AP, Q, q^0, \{f\}, \delta'\rangle$ où*

$$
\begin{aligned}
Q' &= Q \times [\![0, r-1]\!] \\
\delta' &= \{((s,c), p, acc_c(a), (d, next_c(a))) \mid (s,p,a,d) \in \delta,\ c \in [\![0, r-1]\!]\} \\
next_c(a) &= c + \max\{n \in [\![0,r]\!] \mid \forall k \in [\![0,n[\![,\ f_{(c+k) \bmod r} \in a\} \\
acc_c(a) &= \begin{cases} \{f\} & \text{si } next_c(A) < c \text{ ou } a = \mathcal{F} \\ \emptyset & \text{sinon} \end{cases}
\end{aligned}
$$

accepte le même langage que A. Nous dirons que A' est l'automate dégénéralisé de A.

Cette construction, illustrée par la figure 3.10, prend en compte l'optimisation citée à la fin de la section 3.3.3 qui consiste à franchir plusieurs copie d'automate d'un coup. Par exemple si l'on se trouve dans la copie c de l'automate, et que la transition considérée est étiquetée par les conditions d'acceptation $f_c, f_{c+1}, \ldots, f_{r-1}, f_0, f_1$, alors la copie de destination sera la 2. La fonction $next_c$ définie ci-dessus calcule le numéro de la copie suivante en fonction du numéro de la copie courante (c) et des conditions d'acceptation qui étiquettent la transition (a). Les transitions de l'automate A' qui correspondent à des franchissements de f_{r-1} dans A sont marquées comme acceptantes grâce à la fonction acc_c: on sait que f_{r-1} a été franchie si $next_c(a) < c$, mais il est aussi possible qu'une transition soit étiquetée par l'ensemble $\mathcal{F}$ des conditions d'acceptation (dans ce cas $next_c(a) = c$ mais la transition est quand même acceptante).

Déterminisation

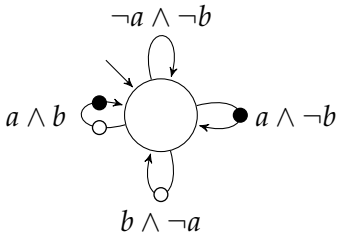


FIG. 3.11: TGBA déterministe acceptant $\mathscr{L}_{\{a,b\}}(\mathsf{GF}\,a \wedge \mathsf{GF}\,b)$. À comparer à la version non-déterministe de la figure 3.7 page 50.

Définition 35. *Un automate de Büchi est déterministe si une séquence de Σ^ω acceptée ne correspond qu'à un unique chemin acceptant de l'automate.*

Dans un TGBA, cela signifie que les ensembles de propositions qui étiquettent les transitions sortantes d'un état doivent être deux à deux disjoints.

Exemple 15. *L'automate de la figure 3.7 page 50 acceptant le même langage que la formule LTL $\mathsf{GF}\,a \wedge \mathsf{GF}\,b$ est non-déterministe: n'importe quel chemin acceptant de l'automate accepte la séquence $(ab)^\omega$.*

Si l'on effectue un produit synchronisé entre un automate A et l'automate de la figure 3.7, les transitions de l'automate A seront dupliquées trois fois si elles vérifient ab, deux fois si elles vérifient $\bar{a}b$ ou $a\bar{b}$, et une fois si elles vérifient $\bar{a}\bar{b}$.

L'automate de la figure 3.11 est déterministe et reconnaît le même langage.

Tout automate de Büchi n'est pas déterminisable. Par exemple il n'existe pas d'automate de Büchi déterministe reconnaissant le langage $(A+B)^* \cdot B^\omega \subset \{A,B\}^\omega$ [136, prop. 8].

Dans les sections 4.3.1 page 91 et 4.3.2 page 93 nous étudierons comment favoriser la production d'automates déterministes lors de la traduction de formules LTL. En particulier, nous serons capable de produire directement l'automate de la figure 3.11.

3.3.7 Vérification des séquences finies

La raison pour laquelle nous travaillons sur des systèmes réactifs et que nous ayons insisté sur le fait qu'ils ne terminent pas (par exemple pages 13 et 19), est que, dans le produit synchronisé, nous ne sommes pas en mesure de distinguer si une exécution est finie du fait du système ou du fait de la synchronisation avec l'automate de la formule.

Si l'on souhaite vérifier des systèmes dont les exécutions ne sont pas forcément infinies, il est nécessaire de modifier l'approche.

La figure 3.12 page suivante illustre le problème sur un exemple simpliste. La structure de Kripke de la figure 3.12a présente deux branches: celle de gauche est finie et vérifie a dans tous ses états, celle de droite est infinie et ne vérifie a que dans son premier état. Selon la sémantique que nous avons donnée à LTL (définition 14 page 38) ce système vérifie la formule $\mathsf{F}\,\neg a$. En effet, la sémantique n'est définie que sur les séquences infinies, et la seule exécution infinie du système est ici la branche de droite de l'automate,

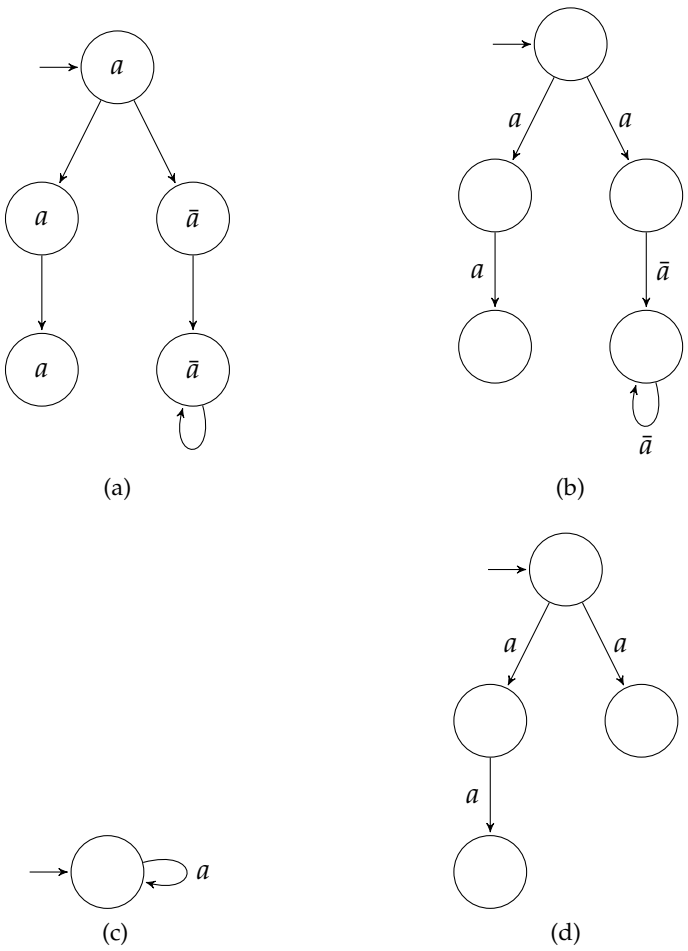


FIG. 3.12: Tentative de vérification de la formule $\mathsf{F}\neg a$ sur un système possédant une branche infinie et une branche finie. (a) structure de Kripke représentant le modèle à vérifier ; (b) son interprétation sous forme de TGBA ; (c) automate acceptant la négation de la formule $\mathsf{F}\neg a$; (d) produit synchronisé de (b) et (c) ne possédant aucun chemin infini. Aucun des TGBA présentés ici ne possède de condition d'acceptation.

qui vérifie $\mathsf{F}\,\neg a$ sans difficulté. Si nous appliquons l'approche automate, c'est-à-dire que nous traduisons la structure de Kripke et la négation de la formule LTL sous forme de TGBA (figure 3.12b et 3.12c) et que nous exécutons un *emptiness check* sur leur produit (figure 3.12d), le résultat est cohérent: le langage de l'automate de la figure 3.12d est vide car le système vérifie la formule.

Des deux branches de la figure 3.12d, l'une est finie parce qu'elle l'était dans le modèle d'origine, l'autre est finie à cause du produit synchronisé. Parce que nous devons ignorer ces dernières, nous ignorons aussi les premières.

Sur les systèmes où l'on souhaite vérifier aussi les séquences finies, une approche consiste à utiliser un type d'automate hybride entre automate de Büchi et automate fini appelé *testeur* [133]. Nous proposons une approche différente qui permet de réutiliser tous les algorithmes de l'approche automate (traduction de formule LTL en automates, produit synchronisés, *emptiness checks*) sans changement. Ce sont les automates représentant le modèle et la formule LTL qui vont subir de légères modifications.

Commençons par redéfinir la sémantique de LTL de la définition 14 page 38 afin de couvrir les séquences finies.

Définition 36. *La satisfaction d'une formule LTL f par rapport à une séquence $\sigma \in (2^{AP})^\infty$ est notée $\sigma \models f$ et définie inductivement de la façon suivante. (Rappelons que $|\sigma| = \omega$ si la séquence est infinie, et que $\omega + 1 = \omega$.) Pour toute proposition atomique p et toutes formules LTL f_1 et f_2:*

$$
\begin{array}{ll}
\sigma \models p & \textit{ssi } |\sigma| > 0 \textit{ et } p \in \sigma(0) \\
\sigma \models \neg f_1 & \textit{ssi } \neg(\sigma \models f_1) \\
\sigma \models f_1 \wedge f_2 & \textit{ssi } \sigma \models f_1 \textit{ et } \sigma \models f_2 \\
\sigma \models \mathsf{X}\, f_1 & \textit{ssi } |\sigma| > 1 \textit{ et } \sigma^1 \models f_1 \\
\sigma \models f_1 \,\mathsf{U}\, f_2 & \textit{ssi } \exists i \in [\![0, |\sigma| + 1[\![\textit{ tel que } \sigma^i \models f_2 \textit{ et } \forall j \in [\![0, i-1]\!],\ \sigma^j \models f_1
\end{array}
$$

Les changements portent sur l'interprétation des opérateurs X et U. L'opérateur X ne peut être vrai en fin de séquence. L'opérande de droite de l'opérateur U doit être vérifié avant la fin de la séquence.

Pour vérifier cette sémantique nous modifions la transformation de la structure de Kripke en TGBA de deux façons:
- nous ajoutons une nouvelle variable propositionnelle, nommée *dead*, qui n'est vraie que sur les états sans successeur,
- chaque état sans successeur se voit équipé d'une boucle sur lui-même.

Plus formellement, une structure de Kripke $\langle AP, \mathcal{Q}, q^0, \delta, l\rangle$ sera interprétée comme le TGBA $\langle 2^{AP\cup\{dead\}}, \mathcal{Q}, \{q^0\}, \emptyset, \delta'\rangle$ où

$$\begin{aligned}\delta' = \big\{(t^{\text{in}}, t^{\text{prop}}, \emptyset, t^{\text{out}}) \in \mathcal{Q}\times 2^{\Sigma}\times\{\emptyset\}\times\mathcal{Q} \mid & t^{\text{out}}\in\delta(t^{\text{in}}),\\ & t^{\text{prop}} = l(t^{\text{in}})\cup\{\overline{dead}\}, \text{ et}\\ & t^{\text{acc}} = \{f\in\mathcal{F}\mid t^{\text{in}}\in f\}\big\}\\ \cup\big\{(t^{\text{in}}, t^{\text{prop}}, \emptyset, t^{\text{out}}) \in \mathcal{Q}\times 2^{\Sigma}\times\{\emptyset\}\times\mathcal{Q} \mid & \delta(t^{\text{in}})=\emptyset,\\ & t^{\text{prop}} = l(t^{\text{in}})\cup\{dead\}, \text{ et}\\ & t^{\text{acc}} = \{f\in\mathcal{F}\mid t^{\text{in}}\in f\}\big\}\end{aligned}$$

Par la suite, nous noterons M le TGBA représentant la structure de Kripke sans ajout des propositions *dead* et sans les boucles, et M' le TGBA défini ci-dessus. Une exécution infinie $\sigma\in\mathscr{L}(M)$ de M correspond à une exécution infinie $\sigma'\in\mathscr{L}(M')$ de M' telle que $\sigma' = (\sigma \parallel \overline{dead}^{\omega})$. Une exécution finie maximale $\sigma = \sigma(0)\sigma(1)\cdots\sigma(n)\in\mathscr{L}(M)$ de M correspond à l'exécution infinie $\sigma'\in\mathscr{L}(M')$ de M' définie par

$$\sigma' = \sigma(0)\overline{dead}\cdot\sigma(1)\overline{dead}\cdots\sigma(n-1)\overline{dead}\cdots(\sigma(n)dead)^{\omega} \tag{3.2}$$

Au lieu de vérifier si $M\models\varphi$, nous proposons de vérifier si $M'\models\varphi'$ avec φ' définie à partir de φ de la façon suivante. (Les opérateurs dérivés sont réécrits en conséquence.)

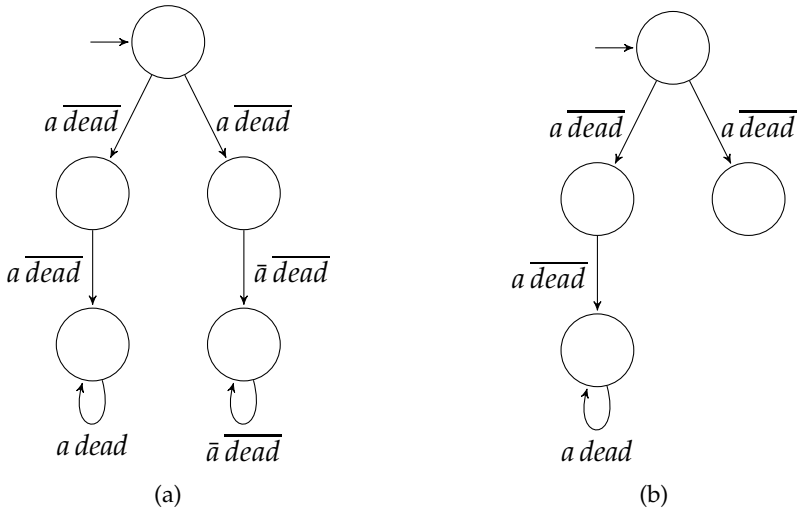
$$\begin{aligned} p' &= p && \text{si } p\in AP\\ (\neg f)' &= \neg f'\\ (f\wedge g)' &= f'\wedge g'\\ (\mathsf{X}\,g)' &= \overline{dead}\wedge\mathsf{X}\,g'\\ (f\,\mathsf{U}\,g)' &= f'\,\mathsf{U}\,g'\end{aligned} \tag{3.3}$$

La seule modification est donc l'ajout d'une contrainte $\overline{dead}$ devant chaque opérateur X. En effet la sémantique de X sur les séquences finies impose que l'état possède effectivement un successeur dans la structure de Kripke d'origine. Il peut sembler surprenant que l'opérateur U ne demande aucune modification sur la formule alors que sa sémantique a aussi été changée pour les séquences finies ; en réalité l'ajout des boucles dans le TGBA suffit à satisfaire cette nouvelle sémantique.

La figure 3.13a page suivante montre la structure de Kripke de la figure 3.12a page 58 interprétée de cette façon. Comme la formule LTL que nous vérifions ne possède pas de X, elle n'a pas besoin d'être réécrite. Après produit synchronisé avec l'automate de la figure 3.12c nous obtenons l'automate de la figure 3.13b page suivante qui contient un chemin acceptant (la branche de gauche est infinie).

Proposition 11. *Avec les notations ci-dessus, $M\models\varphi \iff M'\models\varphi'$ avec la sémantique de la définition 36 page précédente.*

Démonstration. Pour une séquence $\sigma\in\mathscr{L}(M)$ et la séquence correspondante $\sigma'\in\mathscr{L}(M')$ définie par l'équation (3.2), nous montrons $\sigma\models\varphi \iff \sigma'\models\varphi'$ par induction sur l'ensemble des formules LTL.



(a) (b)

FIG. 3.13: Vérification de la formule $\mathsf{F}\neg a$ avec la nouvelle interprétation. (a) nouvelle interprétation sous forme de TGBA de la structure de Kripke de la figure 3.12a page 58 ; (b) produit synchronisé de (a) et de l'automate de la figure 3.12c. Aucun des TGBA présentés ici ne possède de condition d'acceptation.

Si φ est une proposition atomique, $\sigma \models \varphi \iff \sigma' \models \varphi'$ de façon évidente.

Considérons maintenant deux formules f_1 et f_2 pour lesquelles on suppose que $\forall i \in \{1,2\}, \forall \sigma \in \mathscr{L}(M), \sigma \models f_i \iff \sigma' \models f_i'$.

Alors nous avons $\sigma \models \neg f_1 \iff \sigma' \models (\neg f_1)'$ et $\sigma \models f_1 \wedge f_2 \iff \sigma' \models (f_1 \wedge f_2)'$ de façon triviale.

Si $\sigma \models \mathsf{X} f_1$ alors $|\sigma| > 1 \wedge \sigma^1 \models f_1$, on en déduit que $\mathit{dead} \notin \sigma'(0)$ (d'après la définition de σ') et $\sigma'^1 \models f_1'$ (d'après l'hypothèse d'induction). D'autre part, si $\sigma \not\models \mathsf{X} f_1$ alors
– soit $|\sigma| = 1$, on en déduit $\mathit{dead} \in \sigma'(0)$, donc $\sigma' \not\models \mathit{dead} \wedge \mathsf{X} f_1$;
– soit $\sigma^1 \not\models f_1$ alors $\sigma' \not\models \mathit{dead} \wedge \mathsf{X} f_1$.
Nous avons donc $\sigma \models \mathsf{X} f_1 \iff \sigma' \models (\mathsf{X} f_1)'$.

Considérons enfin l'opérateur U. Si σ est une séquence infinie, la sémantique de l'opérateur est inchangée et il est évident que nous avons $\sigma \models f_1 \mathsf{U} f_2 \iff \sigma' \models f_1' \mathsf{U} f_2'$ par induction. Limitons-nous donc au cas où la séquence σ est finie. Si $\sigma \models f_1 \mathsf{U} f_2$, alors $\exists i \in [\![0, |\sigma|+1[\![$ tel que $\sigma^i \models f_2$ et $\forall j \in [\![0, i-1]\!]$, $\sigma^j \models f_1$. Avec l'hypothèse d'induction on a $\exists i \in [\![0, |\sigma|+1[\![$ tel que $\sigma'^i \models f_2'$ et $\forall j \in [\![0, i-1]\!]$, $\sigma'^j \models f_1'$, ce qui suffit pour établir $\sigma' \models f_1' \mathsf{U} f_2'$. Nous avons donc prouvé que $\sigma \models f_1 \mathsf{U} f_2 \implies \sigma' \models f_1' \mathsf{U} f_2'$ L'autre sens est moins évident. Nous prouvons que $\sigma \not\models f_1 \mathsf{U} f_2 \implies \sigma' \not\models f_1' \mathsf{U} f_2'$ par l'absurde en supposant qu'il existe une séquence finie σ telle que $(\sigma \not\models f_1 \mathsf{U} f_2) \wedge (\sigma' \models f_1' \mathsf{U} f_2')$. Par définition nous avons donc

$$\exists i \in [\![0, \omega[\![, \sigma'^i \models f_2' \text{ et } \forall j \in [\![0, i-1]\!], \sigma'^j \models f_1' \tag{3.4}$$

$$\nexists i \in [\![0, |n|]\!], \sigma^i \models f_2 \text{ et } \forall j \in [\![0, i-1]\!], \sigma^j \models f_1 \tag{3.5}$$

D'après l'hypothèse d'induction, on peut réécrire l'équation (3.5) sous la forme

$$\nexists i \in [\![0, |n|]\!], \sigma'^i \models f_2' \text{ et } \forall j \in [\![0, i-1]\!], \sigma'^j \models f_1' \tag{3.6}$$

Des équations (3.4) et (3.6) nous déduisons que le i ne peut pas être pris dans l'intervalle $[\![0, |n|]\!]$. La première équation se réécrit alors

$$\exists i \in]\!]n, \omega[\![, \sigma'^i \models f_2 \text{ et } \forall j \in [\![0, i-1]\!], \sigma'^j \models f_1 \tag{3.7}$$

D'autre part, en poussant la négation de l'équation (3.6) nous obtenons

$$\forall i \in [\![0, |n|]\!], \sigma^i \not\models f_2 \text{ ou } \exists j \in [\![0, i-1]\!], \sigma^j \not\models f_1 \tag{3.8}$$

Nous savons par l'équation (3.6) que la dernière partie de l'équation (3.7) est impossible: $\sigma^j \models f_1$ pour tout $j \leq n$. L'équation (3.7) se simplifie donc en

$$\forall i \in [\![0, |n|]\!], \sigma^i \not\models f_2 \tag{3.9}$$

Comme $\sigma'^n \not\models f_2$ d'après l'équation 3.9 et que la séquence σ' se répète à partir de l'indice n, nous en déduisons que $\forall i \in [\![n, \omega[\![, \sigma'^i \not\models f_2$, ce qui est en contradiction avec l'équation (3.7). Nous avons donc prouvé que $\sigma \models f_1 \,\mathsf{U}\, f_2 \iff \sigma' \models f_1' \,\mathsf{U}\, f_2'$.

Par induction sur l'ensemble des formules LTL, nous avons donc $\sigma \models \varphi \iff \sigma' \models \varphi'$ pour toute séquence σ finie ou infinie, et toute formule LTL φ. □

Remarques. La nouvelle proposition *dead* peut être utilisée explicitement dans les formules LTL, par exemple pour prouver que le système se termine après qu'une condition soit rencontrée.

$$\mathsf{G}(\mathit{condition} \rightarrow \mathsf{F}\, \mathit{dead})$$

Pour des scénarii où le modèle serait composé par la synchronisation de plusieurs automates nous pourrions aussi imaginer plusieurs propositions $\mathit{dead}_1, \mathit{dead}_2, \ldots$ pour chacun des sous-systèmes.

3.4 Conclusion

Dans ce chapitre nous avons introduit les automates de Büchi généralisés étiquetés sur les transitions (TGBA). À travers les propositions 3 et 4 (page 51), nous avons montré qu'ils étaient plus concis que les automates de Büchi généralisés étiquetés sur les états.

Ces TGBA ont déjà été utilisés par divers auteurs [96, 34, 58, 63, 123] parce qu'ils rendent la traduction d'une formule LTL plus naturelle, mais rares sont ceux qui ont poursuivi l'approche automate avec ces TGBA jusqu'au bout [34, 123]. L'habitude de ramener la vérification à des automates de Büchi classiques perdure.

Dans le chapitre suivant nous aborderons la traduction de formules LTL en TGBA. L'explication de la méthode des tableaux montrera de façon assez frappante l'intérêt d'étiqueter les automates sur les transitions.

Dans le chapitre 5 nous listerons les différents algorithmes d'*emptiness check* existant pour décider de la vacuité d'un TGBA. Nous comparerons l'approche basée sur les TGBA à celle basée sur les automates de Büchi classiques. C'est ici plus l'importance de la *généralisation* des conditions d'acceptation qui sera mise en avant.

À notre avis le principal intérêt des TGBA est illustré par la figure 3.11 page 57. Les formules du type $\mathsf{GF}\,a \wedge \mathsf{GF}\,b$ sont des formules d'*équité faible*. Le fait que n'importe quelle formule du type $\bigwedge_i \mathsf{GF}\,p_i$ puisse être représentée par un TGBA déterministe à un seul état signifie qu'une approche du *model checking* basée sur les TGBA (et utilisant un algorithme d'*emptiness check* insensible au nombre de conditions d'acceptation) peut prendre en compte des hypothèses d'équité faible sans aucun surcoût. Nous reviendrons sur ce point dans le chapitre 7.

Chapitre 4

Traduction de formules LTL

Ce chapitre introduit différentes techniques de traduction de formules LTL en TGBA. Il se concentre ensuite sur l'une d'entre elles pour proposer plusieurs optimisations visant à réduire la taille de l'automate produit et ceci pour accélérer le processus de vérification.

4.1 Existant

La figure 4.1 pages 66–67 présente une bibliographie des algorithmes de traduction de formule LTL en automate de Büchi, ainsi que certains travaux fondateurs (mais pas directement concernés par la traduction). Les flèches pleines représentent la filiation principale de l'algorithme présenté tandis que les pointillés indiquent des reprises d'idées.

L'ensemble de ces algorithmes peut être abordé selon deux angles:
- Le type d'automate construit.
  Les anciens algorithmes cherchent pour la plupart à construire un automate de Büchi (non généralisé) étiqueté sur les états, tandis que les plus récents produisent des TGBA.
- la famille de construction.
  Nous pouvons distinguer quatre types de méthodes:
  - les constructions combinatoires
  - les constructions par composition
  - les constructions par tableau
  - les constructions par automates alternants

Les prochaines sections sont dédiées à chacune de ces familles de constructions.

4.1.1 Constructions combinatoires

La première traduction d'une formule φ vers un automate de Büchi fut proposée par Wolper et al. [148], dans un cadre non spécifique à LTL. Cette traduction *produit un automate dont le nombre d'états est systématiquement de l'ordre de* $2^{O(|\varphi|)}$ où $|\varphi|$ représente la taille d'une formule LTL.

Il s'agit d'un algorithme dont la correction est facile à établir mais dont la complexité est toujours celle du pire cas. Aussi nous ne nous y attarderons pas.

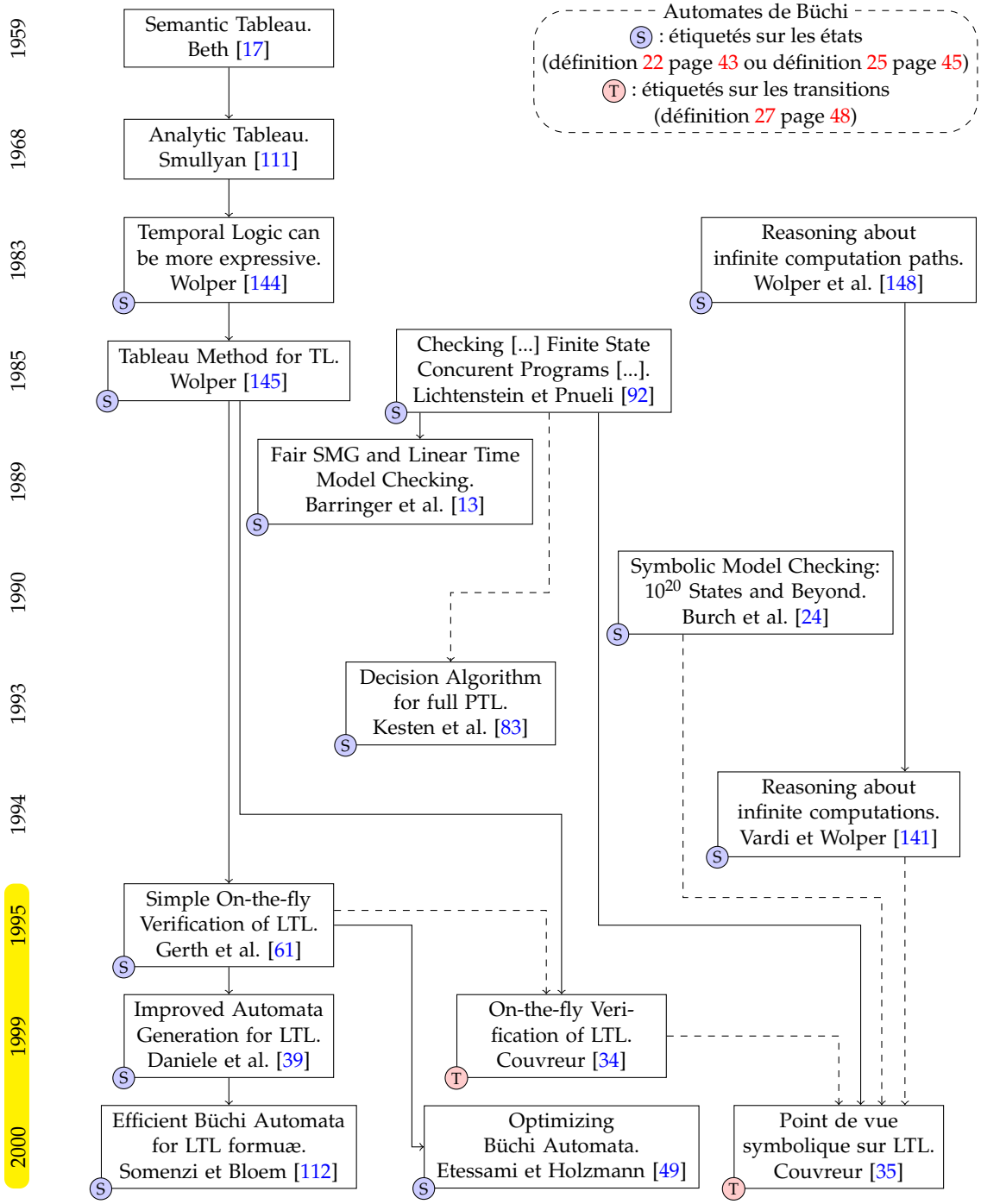


FIG. 4.1: Généalogie partielle des algorithmes de traduction de formule LTL en automate de Büchi.

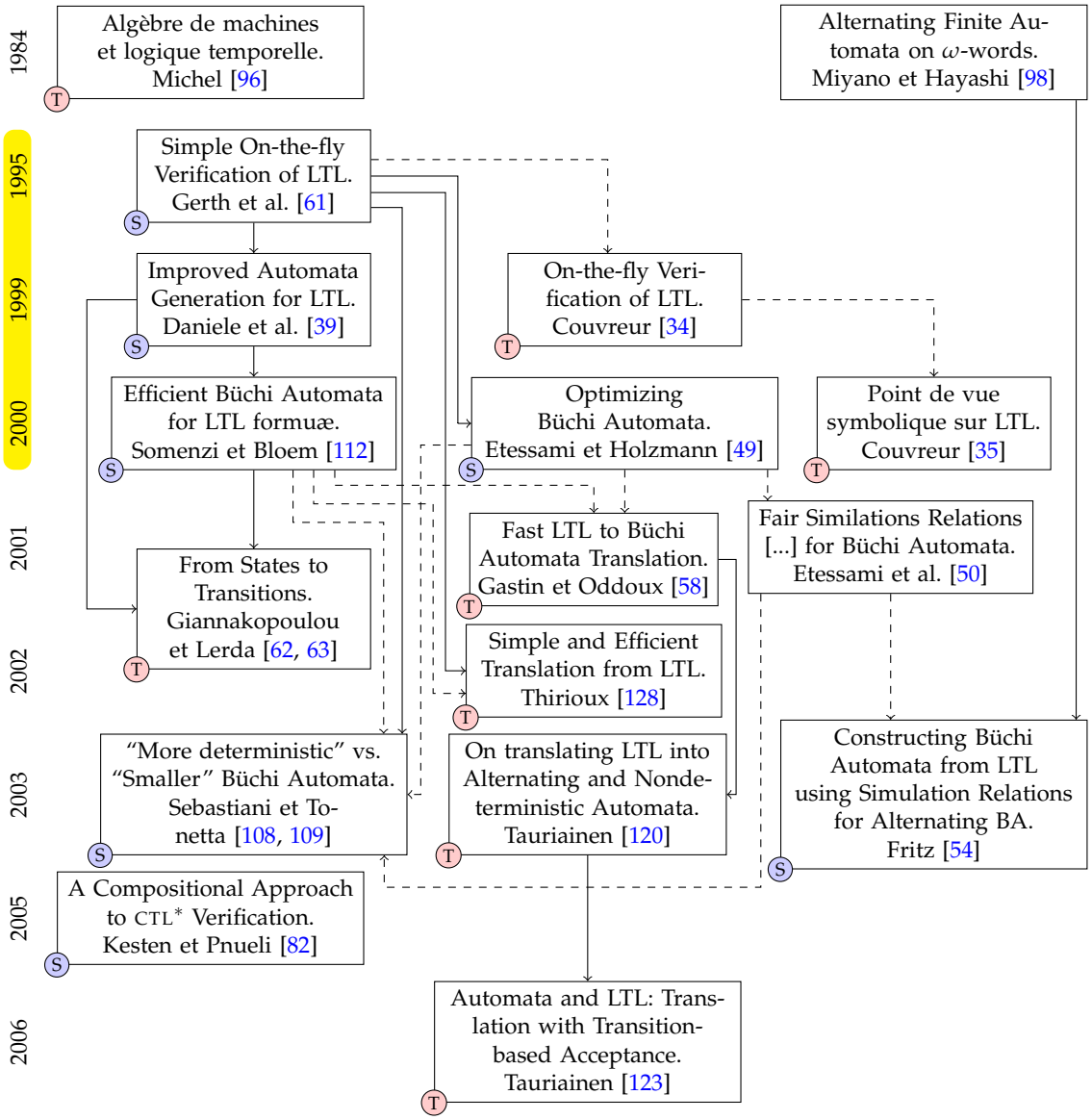

1984
1995
1999
2000
2001
2002
2003
2005
2006
Algèbre de machines et logique temporelle. Michel [96]
Alternating Finite Automata on ω-words. Miyano et Hayashi [98]
Simple On-the-fly Verification of LTL. Gerth et al. [61]
Improved Automata Generation for LTL. Daniele et al. [39]
On-the-fly Verification of LTL. Couvreur [34]
Efficient Büchi Automata for LTL formuæ. Somenzi et Bloem [112]
Optimizing Büchi Automata. Etessami et Holzmann [49]
Point de vue symbolique sur LTL. Couvreur [35]
Fast LTL to Büchi Automata Translation. Gastin et Oddoux [58]
Fair Similations Relations [...] for Büchi Automata. Etessami et al. [50]
From States to Transitions. Giannakopoulou et Lerda [62, 63]
Simple and Efficient Translation from LTL. Thirioux [128]
"More deterministic" vs. "Smaller" Büchi Automata. Sebastiani et Tonetta [108, 109]
On translating LTL into Alternating and Nondeterministic Automata. Tauriainen [120]
Constructing Büchi Automata from LTL using Simulation Relations for Alternating BA. Fritz [54]
A Compositional Approach to CTL* Verification. Kesten et Pnueli [82]
Automata and LTL: Translation with Transition-based Acceptance. Tauriainen [123]

FIG. 4.1: Suite.

La traduction repose sur la définition d'un ensemble $cl(\varphi)$ (*cl* pour *closure*) contenant l'ensemble des sous-formules de φ, ainsi que leurs négations.

Par exemple $cl(p_1 \vee \mathsf{X}\, p_2) = \{p_1, \neg p_1, p_2, \neg p_2, \mathsf{X}\, p_2, \neg \mathsf{X}\, p_2, p_1 \vee \mathsf{X}\, p_2, \neg(p_1 \vee \mathsf{X}\, p_2)\}$.

Cet ensemble peut être construit simplement, et l'on montre que $|cl(\varphi)| \leq 2|\varphi|$.

A_φ est construit en créant un état pour chaque sous ensemble de $cl(\varphi)$ contenant des formules non contradictoires. Cela fait donc moins de $2^{|cl(\varphi)|}$ états car les états incohérents, comme ceux contenant à la fois p et $\neg p$, sont supprimés.

Les transitions entre ces états sont posées de façon à respecter l'interprétation des formules logiques. Par exemple l'état $\{p_1, \mathsf{X}\, p_2\}$ ne peut être relié qu'à des états contenant p_2. Les transitions sont étiquetées par l'ensemble des littéraux apparaissant dans l'état source. (L'automate ainsi produit est équivalent à un automate étiqueté sur les états, car toutes les transitions sortantes d'un état sont étiquetées pareillement.)

Les états initiaux sont les états dont l'étiquette contient φ.

Enfin, les conditions d'acceptation sont définies de façon à nous assurer que les sous-formules du type $a \,\mathsf{U}\, b$ n'acceptent pas de séquences vérifiant a infiniment sans jamais vérifier b. Il suffit pour cela d'imposer que tout chemin passe infiniment par un état où $a \,\mathsf{U}\, b$ et b apparaissent ensemble, ou par un état où $a \,\mathsf{U}\, b$ n'apparaît pas. Ceci peut être exprimé sous la forme d'une condition d'acceptation de Büchi généralisée :

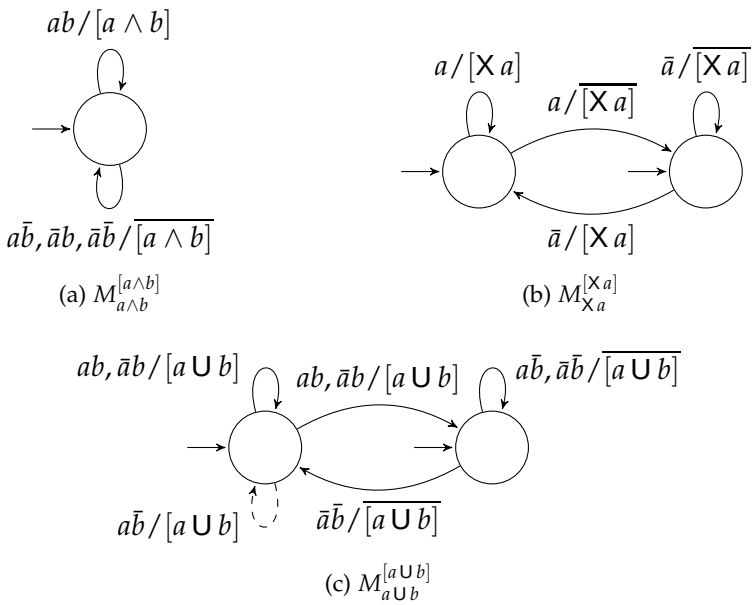
$$\mathcal{F} = \bigcup_{(a \,\mathsf{U}\, b) \in cl(\varphi)} \{\mathcal{F}_{a \,\mathsf{U}\, b}\} \quad \text{où} \quad \mathcal{F}_{a \,\mathsf{U}\, b} = \{s \in S \mid b \in s \vee (a \,\mathsf{U}\, b) \notin s\} \tag{4.1}$$

Wolper et al. [148], qui considèrent des automates de Büchi avec un seul ensemble d'états d'acceptation, expriment ces multiples conditions d'acceptation sous la forme d'un second automate, synchronisé par la suite avec celui de la formule. Wolper [146] présente cet algorithme dans un cadre restreint à LTL, et en construisant un automate de Büchi généralisé.

Couvreur [35] propose une traduction similaire tirant partie des BDD (diagrammes de décision binaires). L'utilisation des BDD permet de représenter la structure de l'automate de façon entièrement symbolique, sans devoir expliciter ses états et transitions. Cette construction n'est appropriée que pour des algorithmes de vérification capables de raisonner sur cette structure symbolique, par exemple en utilisant des points fixes. Comme nous le verrons dans les mesures de la section 4.4 page 98, il y a peu d'intérêt à utiliser cette construction pour explorer la structure de l'automate explicitement.

4.1.2 Constructions par composition

En 1984, Michel [96] introduisait une traduction de LTL basée sur la composition de transducteurs comparables à des machines de Mealy mais équipés de conditions d'acceptation pour accepter des mots infinis. La structure finalement obtenue après composition est comparable à un TGBA mais en utilisant les conditions d'acceptation duales: Michel appelle *graphe instable* un ensemble de transitions dans lesquelles une exécution ne doit pas rester bloquée. Nous représenterons ces transitions en pointillés.



(a) $M_{a\wedge b}^{[a\wedge b]}$ (b) $M_{\mathsf{X}\, a}^{[\mathsf{X}\, a]}$

(c) $M_{a\,\mathsf{U}\, b}^{[a\,\mathsf{U}\, b]}$

FIG. 4.2: Transducteurs correspondant aux fonctions $T_{a\wedge b}^{[a\wedge b]}$, $T_{\mathsf{X}\, a}^{[\mathsf{X}\, a]}$ et $T_{a\,\mathsf{U}\, b}^{[a\,\mathsf{U}\, b]}$.

L'idée est qu'une formule $\varphi \in \mathrm{LTL}_{AP}$ peut être associée à une fonction $T_\varphi^{[\varphi]}$ qui pour un ω-mot $\sigma \in (2^{AP})^\omega$ retourne un ω-mot $\sigma' = T_\varphi^{[\varphi]}(\sigma) \in (2^{\{[\varphi]\}})^\omega$ où

$$\forall i \geq 0, \quad \sigma'(i) = \begin{cases} \{[\varphi]\} & \text{si } \sigma^i \models \varphi \\ \{\} & \text{si } \sigma^i \not\models \varphi \end{cases}$$

c'est-à-dire que $\sigma'(i)$ indique si σ valide la formule φ à l'instant i. Dans ces notations, l'indice φ de la fonction $T_\varphi^{[\varphi]}$ indique la formule reconnue par cette fonction, tandis que l'exposant $[\varphi]$ indique le symbole émis lorsque le formule est vraie. Lorsque nous composerons ces fonctions par la suite, il nous sera utile de pouvoir choisir le symbole émis.

Exemple 16. *En utilisant les abréviations de la section 3.1.4 page 34 (c'est-à-dire $ab \equiv \{a, b\}$, $a\bar{b} \equiv \{a\}$, etc.) si l'on a*

$$\sigma = \left(\bar{a}b \cdot a\bar{b} \cdot ab \cdot a\bar{b} \cdot \bar{a}\bar{b}\right)^\omega \in \left(2^{\{a,b\}}\right)^\omega$$

alors par exemple

$$T_{a\,\mathsf{U}\, b}^{[a\,\mathsf{U}\, b]}(\sigma) = \left([a \mathsf{U} b] \cdot [a \mathsf{U} b] \cdot [a \mathsf{U} b] \cdot \overline{[a \mathsf{U} b]} \cdot \overline{[a \mathsf{U} b]}\right)^\omega \in \left(2^{\{[a\,\mathsf{U}\, b]\}}\right)^\omega$$

$[a \mathsf{U} b]$ doit ici être compris comme un symbole indiquant que la formule $a \mathsf{U} b$ est vraie à l'instant correspondant, tandis que $\overline{[a \mathsf{U} b]}$ indique le contraire.

Ce type de fonction peut être exprimé par les transducteurs non-déterministes introduits par Michel [96]. La figure 4.2 en donne trois exemples. Les étiquettes de transitions indiquent les symboles lus par l'automate à gauche du « / » puis le symbole produit à

droite. Ces automates définissent une relation entre deux langages: une paire d'ω-mots (σ, σ') est dans la relation s'il existe une exécution infinie de l'automate qui lit σ et produit σ' sans utiliser de transition en pointillés de façon continue (c'est-à-dire sans rester bloqué dans le *graphe instable*).

Le transducteur $M_{a\wedge b}^{[a\wedge b]}$ est le plus simple, car il est déterministe. Il prend en entrée un ω-mot de $(2^{a,b})^\omega$, et produit un ω-mot de $(2^{\{a\wedge b\}})^\omega$ en sortie: si le symbole ab est lu, le symbole $[a \wedge b]$ est produit, dans les autres cas le symbole $\overline{[a \wedge b]}$ est produit. Ce transducteur correspond bien à l'idée qu'on pourrait se faire d'une porte logique avec deux lignes en entrée (les valeurs de vérité de a et b) et une sortie (la valeur de vérité de $a \wedge b$).

Le transducteur $M_{\mathsf{X}a}^{[\mathsf{X}a]}$ introduit le besoin d'indéterminisme. En effet, à un instant donné, il n'est pas possible de connaître la valeur de vérité de l'instant suivant. Produire la valeur de vérité de $\mathsf{X}\,a$ demande donc de faire un pari sur l'avenir. Comme nous travaillons sur des exécutions infinies, il suffit de s'arranger pour qu'une exécution s'interrompe dès qu'on s'aperçoit qu'un choix était mauvais. Les deux états de l'automate correspondent aux deux possibilités:
– chaque fois que l'automate émet $[\mathsf{X}\,a]$, il arrive dans l'état de gauche, à partir duquel l'exécution ne peut se prolonger que si a est effectivement lu ;
– inversement, l'émission de $\overline{[\mathsf{X}\,a]}$ amène systématiquement l'automate dans l'état de droite, à partir duquel l'exécution ne peut se prolonger que si $\bar{a}$ est lu.
L'indéterminisme permet d'envisager les deux choix systématiquement.

Le transducteur $M_{a\,\mathsf{U}\,b}^{[a\,\mathsf{U}\,b]}$ montre comment reconnaître l'opérateur U. (L'automate M_{U} présenté page 291 de l'article d'origine [96] est faux: il émet la séquence $[p\,\mathsf{U}\,q]^\omega$ si l'entrée est $(p\bar{q})^\omega$.) Là encore l'indéterminisme est utilisé pour envisager les deux cas possibles. L'état de gauche correspond au cas où l'automate espère que l'instant courant vérifie $a\,\mathsf{U}\,b$ tandis que celui de droite représente la possibilité $\overline{a\,\mathsf{U}\,b}$. Quand b est vrai, l'automate peut produire $[a\,\mathsf{U}\,b]$ sans danger, et cela n'est possible que dans l'hypothèse de gauche. De la même façon, si $\bar{a}\bar{b}$ est reçu, l'automate peut émettre $\overline{[a\,\mathsf{U}\,b]}$ sans se tromper, et cela seulement à droite. Le dernier cas, $a\bar{b}$, ne permet pas d'infirmer l'une ou l'autre des hypothèses et apparaît de chaque côté. La boucle en pointillés interdit un passage continu: un mot dont le suffixe est $(a\bar{b})^\omega$ (qui, par définition, ne vérifie pas $a\,\mathsf{U}\,b$) ne peut donc être accepté que par la boucle qui émet $\overline{[a\,\mathsf{U}\,b]}$.

Ces trois transducteurs peuvent servir de briques de base pour construire, par composition, un transducteur acceptant n'importe quelle formule LTL donnée sous forme normale positive (définition 16 page 39). Il suffit de considérer les sous-formules LTL comme des propositions atomiques dont la valeur de vérité sera produite par un autre transducteur.

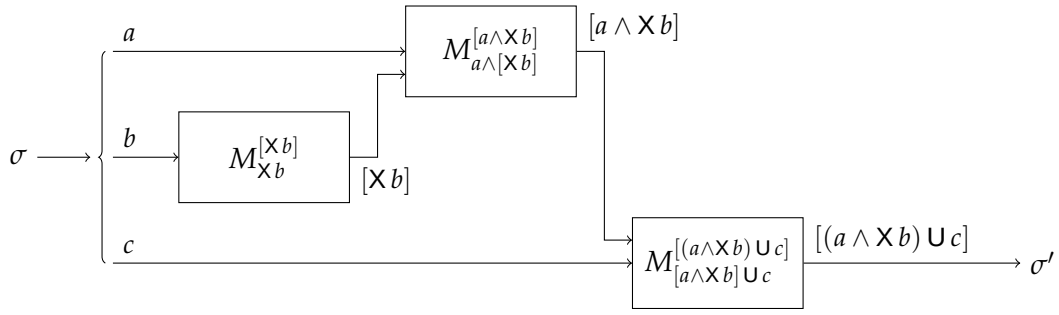


FIG. 4.3: Composition de transducteurs pour $(a \wedge \mathsf{X}\, b)\,\mathsf{U}\, c$.

$$
\begin{aligned}
T_a^{[a]}(\sigma) &= \left(i \mapsto \begin{cases} [a] & \text{si } \sigma_{|\{a\}}(i) = a \\ \overline{[a]} & \text{sinon} \end{cases} \right) && \text{pour toute proposition atomique } a \\
T^{[\varphi\wedge\psi]}_{\varphi\wedge\psi}(\sigma) &= T^{[\varphi\wedge\psi]}_{[\varphi]\wedge[\psi]}(T^{[\varphi]}_{\varphi}(\sigma) \parallel T^{[\psi]}_{\psi}(\sigma)) && \text{pour deux formules LTL } \varphi \text{ et } \psi \\
T^{[\mathsf{X}\,\varphi]}_{\mathsf{X}\,\varphi}(\sigma) &= T^{[\mathsf{X}\,\varphi]}_{\mathsf{X}[\varphi]}(T^{[\varphi]}_{\varphi}(\sigma)) \\
T^{[\varphi\,\mathsf{U}\,\psi]}_{\varphi\,\mathsf{U}\,\psi}(\sigma) &= T^{[\varphi\,\mathsf{U}\,\psi]}_{[\varphi]\,\mathsf{U}\,[\psi]}(T_{\varphi}(\sigma)^{[\varphi]} \parallel T^{[\psi]}_{\psi}(\sigma)) \\
&\text{etc.}
\end{aligned}
$$

Le seul intérêt de la transformation $T_a^{[a]}$ est de projeter σ sur l'alphabet $\{a\}$, la transformation de a en $[a]$ n'est pas indispensable en pratique.

Exemple 17. *Considérons la formule $(a \wedge \mathsf{X}\, b)\,\mathsf{U}\, c$. En appliquant la remarque précédente sur l'emploi de $\sigma_{|\{a\}}$ à la place de $T_a(\sigma)$, nous avons*

$$T^{[(a\wedge \mathsf{X}\, b)\,\mathsf{U}\, c]}_{(a\wedge \mathsf{X}\, b)\,\mathsf{U}\, c}(\sigma) = T^{[(a\wedge \mathsf{X}\, b)\,\mathsf{U}\, c]}_{[a\wedge \mathsf{X}\, b]\,\mathsf{U}\, c}\left(T^{[a\wedge \mathsf{X}\, b]}_{a\wedge[\mathsf{X}\, b]}\left(\sigma_{|\{a\}} \parallel T^{[\mathsf{X}\, b]}_{\mathsf{X}\, b}(\sigma_{|\{b\}})\right) \parallel \sigma_{|\{c\}}\right)$$

Cela suggère qu'un transducteur pour $(a \wedge \mathsf{X}\, b)\,\mathsf{U}\, c$ peut être construit par composition des transducteurs de la figure 4.2 page 69 comme indiqué sur la figure 4.3.

Michel [96] montre comment ces transducteurs peuvent être composés au sens de la figure 4.3 pour obtenir celui de la figure 4.4 page suivante. L'opération est assez intuitive: les nœuds et arcs sont construits par produits cartésiens, en ne retenant que les arcs compatibles (le symbole émis par l'automate de gauche est celui attendu par l'automate de droite).

Michel ne pousse pas sa méthode jusqu'à construire un automate de Büchi (il se sert de cette construction pour relier LTL à la théorie des automates puis établir à nouveau certains résultats à moindre frais). Cependant produire un TGBA acceptant la formule φ à partir du transducteur correspondant demande peu de travail. Ces transducteurs associent un ω-mot $\sigma \in (2^{AP})^\omega$ à un ω-mot $\sigma' \in (2^{\{[\varphi]\}})^\omega$, tel que $\sigma'(i) = [\varphi]$ si et seulement si $\sigma^i \models \varphi$. Pour déterminer uniquement si $\sigma \models \varphi$ nous pouvons donc modifier le transducteur à partir des trois remarques suivantes:

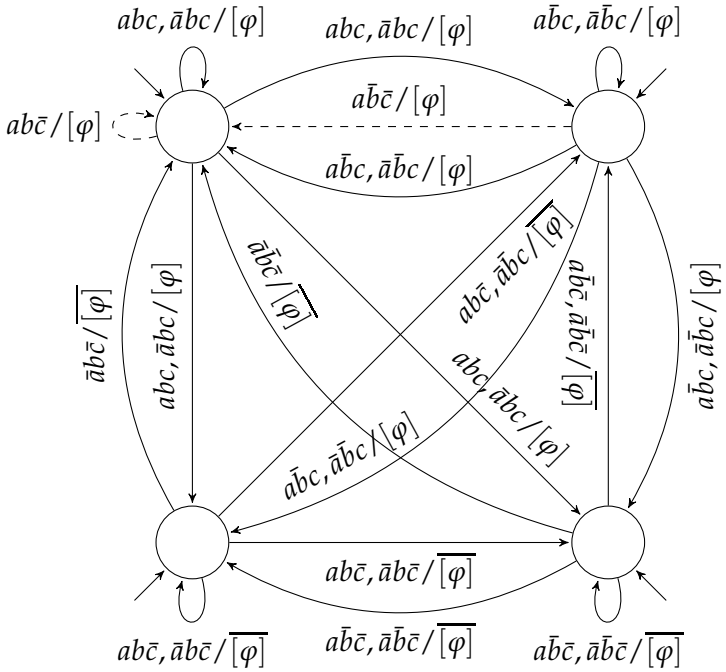


FIG. 4.4: Transducteur pour $\varphi = (a \wedge \mathsf{X}\, b)\, \mathsf{U}\, c$.

– Seuls les chemins qui produisent $[\varphi]$ comme premier symbole nous intéressent.

  Sur l'exemple de la figure 4.4, la modification à apporter est simple car tous les symboles $[\varphi]$ sont émis par les arcs qui quittent les deux états du haut, tandis que les arcs qui quittent les états du bas émettent $\overline{[\varphi]}$. Il suffirait donc de ne garder que les états du haut dans l'ensemble des états initiaux. De cette façon toute exécution acceptée (au sens des *graphes instables*) correspond à un mot validant la formule.

  Dans le cas général, où les arcs quittant un état n'émettent pas tous nécessairement le même symbole, nous pouvons créer un nouvel état initial et y connecter une copie de tous les arcs émettant $[\varphi]$ (dont nous n'aurons changé que l'état source).

– Une fois que l'automate a été modifié comme indiqué ci-dessus pour n'émettre que des mots commençant par $[\varphi]$, nous sommes sûrs qu'un mot n'est accepté par l'automate que s'il vérifie φ. Il est inutile d'émettre la suite de symboles $[\varphi]$ ou $\overline{[\varphi]}$.

– Enfin, nous avons déjà mentionné qu'un *graphe instable* — c'est-à-dire un ensemble des transitions dans lesquelles le transducteur ne doit pas rester continûment — correspondait au dual d'une condition d'acceptation généralisée dans un automate de Büchi étiqueté sur les transitions. De façon générale ces transducteurs peuvent posséder plusieurs graphes instables.

  Pour transformer cette structure en un TGBA, il reste donc à étiqueter les transitions formant le complément de chaque *graphe instable* avec une nouvelle condition d'acceptation. Ce complément peut être fait par rapport à l'automate d'origine, ou à celui équipé des nouvelles transitions quittant l'état initial inséré: comme ces dernières ne

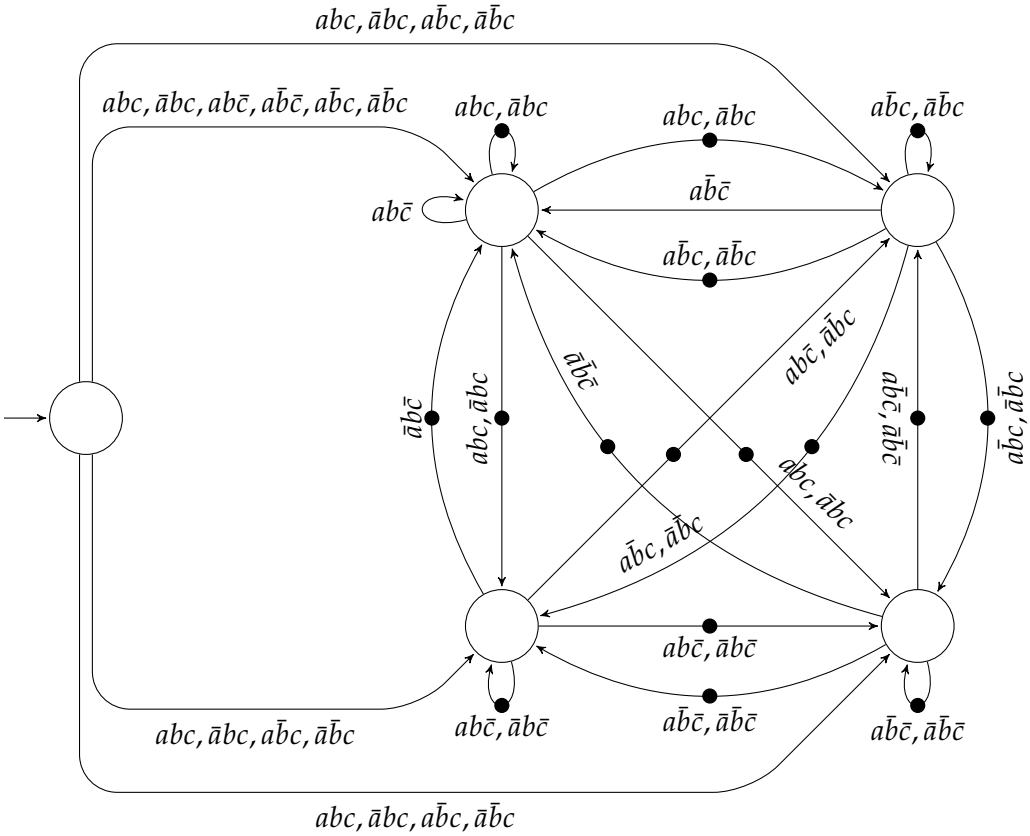


FIG. 4.5: TGBA acceptant $\varphi = (a \wedge \mathsf{X}\, b)\, \mathsf{U}\, c$, construit à partir du transducteur de la figure 4.4 page ci-contre. Il n'y avait qu'un graphe instable dans le transducteur d'origine, nous utilisons donc une seule condition d'acceptation dans ce TGBA: $\mathcal{F} = \{\bullet\}$.

peuvent faire partie d'aucun circuit, cela ne change rien au langage de l'automate (propriété 2 page 51).

La figure 4.5 montre le TGBA obtenu en appliquant ces modifications sur le transducteur de la figure 4.4 page ci-contre.

Nous pouvons dire que cette construction est en fait une version déguisée de l'approche combinatoire. Ici l'explosion est cachée dans l'opérateur de composition des transducteurs: comme chaque transducteur représentant une formule LTL de base ($M_{Xa}^{[Xa]}$ et $M_{a\,\mathsf{U}\,b}^{[a\,\mathsf{U}\,b]}$ figure 4.2 page 69) possède deux états et que l'opérateur de composition est essentiellement un produit cartésien, le transducteur associé à une formule φ est de taille exponentielle par rapport au nombre d'opérateurs LTL. Le cas le plus frappant est celui de la formule $\underbrace{\mathsf{X}\,\mathsf{X}\cdots\mathsf{X}}_{n}\,p$ qui sera traduit en un automate de 2^{n+1} états, alors qu'intuitivement $n+1$ états suffisent.

Une construction similaire, c'est-à-dire construisant un automate par composition de transducteurs, a été réinventée vingt ans plus tard par Kesten et Pnueli [82]. Une illus-

tration claire de cette dernière construction est présentée par Maler et al. [93], section 2. L'inconvénient de cette seconde méthode est qu'elle fait porter les conditions d'acceptation par les états et non les transitions. En conséquence, les automates produits sont plus gros. Par exemple leur version de l'opérateur U utilise 4 états au lieu des deux de la figure 4.2c page 69. Pour revenir sur le lien entre cette classe de construction et la précédente, les formules utilisées par Kesten et Pnueli [82] pour contraindre la relation de transition des automates construits sont les mêmes que celles utilisées par Couvreur [35] dans sa construction combinatoire

4.1.3 Constructions par tableau

La traduction d'une formule LTL en automate de Büchi par la méthode des tableaux vient d'une technique utilisée en logique propositionnelle. Nous commençons par expliquer cette dernière avant de montrer comment elle s'étend à la logique temporelle.

Un *tableau sémantique* est une façon de prouver qu'une formule de logique propositionnelle $f(p_0, p_1, \ldots, p_n)$ est satisfiable[1]. Le plus souvent un tableau est utilisé pour des raisonnements par réfutation, c'est-à-dire que pour prouver que f est une tautologie[2], on montre avec un tableau que $\neg f$ n'est pas satisfiable.

La preuve par tableau prend la forme d'un arbre dont les nœuds sont étiquetés par un ensemble de formules logiques. Dans le cadre de la logique propositionnelle, cet arbre peut être vu comme une mise sous forme normale disjonctive : les feuilles de l'arbre représentent des conjonctions de sous-formules atomiques à satisfaire, tandis que l'arbre lui-même représente la disjonction de ces conjonctions. Une branche peut être vue comme une suite d'implications, des feuilles vers la racine ; c'est-à-dire que la satisfiabilité d'une feuille implique celle de la formule.

Des *règles de tableau* indiquent comment construire le tableau à partir de la formule. Par exemple les 9 premières lignes du tableau[3] 4.1 donnent celles utilisées en logique propositionnelle.

Une preuve par tableau s'effectue en partant de la formule f dont on veut établir la satisfiabilité, puis en appliquant successivement toutes les règles de tableau possibles jusqu'à obtenir soit des nœuds contenant des formules contradictoires (c'est-à-dire contenant à la fois p et $\neg p$, par exemple, ou contenant $\bot$), soit des nœuds irréductibles (plus aucune règle ne s'applique).

Une branche dont l'un des nœuds contient des formules contradictoires est dite *fermée*. La formule f est satisfiable s'il existe une branche non fermée. La formule f n'est pas satisfiable si toutes les branches sont fermées (et dans ce cas, $\neg f$ est un théorème).

[1] C'est-à-dire qu'il existe une affectation A des variables propositionnelles $p_0, p_1, \ldots, p_n$ telle que $f(A(p_0), A(p_1), \ldots, A(p_n)) = \top$.

[2] $f(A(p_0), A(p_1), \ldots, A(p_n)) = \top$ pour toute affectation A.

[3] Le lecteur prendra garde à ne pas confondre tableau et tableau. L'un est un objet mathématique, représenté ici sous forme arborescente, tandis que l'autre est un mode d'organisation où les données sont présentées sous forme de lignes et de colonnes. Par le passé, un tableau avait effectivement la forme d'un tableau [16, 17] avant de prendre progressivement celle d'un arbre [111].

	formule	1er fils	2e fils
(1)	$\Gamma \cup \{\bot\}$	$\{\bot\}$	
(2)	$\Gamma \cup \{\top\}$	Γ	
(3)	$\Gamma \cup \{\neg\top\}$	$\{\bot\}$	
(4)	$\Gamma \cup \{\neg\bot\}$	Γ	
(5)	$\Gamma \cup \{\neg\neg f\}$	$\Gamma \cup \{f\}$	
(6)	$\Gamma \cup \{f \wedge g\}$	$\Gamma \cup \{f, g\}$	
(7)	$\Gamma \cup \{f \vee g\}$	$\Gamma \cup \{f\}$	$\Gamma \cup \{g\}$
(8)	$\Gamma \cup \{\neg(f \wedge g)\}$	$\Gamma \cup \{\neg f\}$	$\Gamma \cup \{\neg g\}$
(9)	$\Gamma \cup \{\neg(f \vee g)\}$	$\Gamma \cup \{\neg f, \neg g\}$	
(10)	$\Gamma \cup \{\neg \mathsf{X} f\}$	$\Gamma \cup \{\mathsf{X} \neg f\}$	
(11)	$\Gamma \cup \{f \,\mathsf{U}\, g\}$	$\Gamma \cup \{g\}$	$\Gamma \cup \{f, \mathsf{X}(f \,\mathsf{U}\, g), \mathsf{P}\, g\}$
(12)	$\Gamma \cup \{\neg(f \,\mathsf{U}\, g)\}$	$\Gamma \cup \{\neg f, \neg g\}$	$\Gamma \cup \{\neg g, \mathsf{X} \neg(f \,\mathsf{U}\, g)\}$
(13)	$\Gamma \cup \{\mathsf{F}\, g\}$	$\Gamma \cup \{g\}$	$\Gamma \cup \{\mathsf{X}\mathsf{F}\, g, \mathsf{P}\, g\}$
(14)	$\Gamma \cup \{\neg(\mathsf{F}\, g)\}$	$\Gamma \cup \{\neg g, \mathsf{X} \neg \mathsf{F}\, g\}$	
(15)	$\Gamma \cup \{f \,\mathsf{R}\, g\}$	$\Gamma \cup \{f, g\}$	$\Gamma \cup \{g, \mathsf{X}(f \,\mathsf{R}\, g)\}$
(16)	$\Gamma \cup \{\neg(f \,\mathsf{R}\, g)\}$	$\Gamma \cup \{\neg g\}$	$\Gamma \cup \{\neg f, \mathsf{X} \neg(f \,\mathsf{R}\, g), \mathsf{P} \neg g\}$
(16)	$\Gamma \cup \{\mathsf{G}\, g\}$	$\Gamma \cup \{g, \mathsf{X}\mathsf{G}\, g\}$	
(17)	$\Gamma \cup \{\neg(\mathsf{G}\, g)\}$	$\Gamma \cup \{\neg g\}$	$\Gamma \cup \{\mathsf{X} \neg \mathsf{G}\, g, \mathsf{P} \neg g\}$

TAB. 4.1: Règles de tableau pour LTL.

La figure 4.6 démontre que $f(A, B, C) = \neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C))$[4] est une tautologie en utilisant un tableau. La racine de l'arbre est un ensemble contenant $\neg f$, la négation de la formule à prouver. L'arbre est construit en appliquant des règles de tableau successivement jusqu'à obtenir une contradiction ou à ne plus pouvoir appliquer de règles de tableau. Dans cet exemple, toutes les branches sont fermées, cela implique que $\neg f$ n'est pas satisfiable, et donc que la formule f est une tautologie.

La méthode prend un peu plus de sens lorsqu'elle est étendue pour considérer les quantificateurs de la logique du premier ordre [17, 16]. Fitting [51] donne la preuve de correction d'une démonstration par tableau, ainsi qu'une implémentation pour la logique propositionnelle.

Ben-Ari et al. [15] étendent ce schéma de preuve aux logiques à temps arborescent, tandis que Wolper [144, 145] s'attaque aux logiques à temps linéaire.

L'idée principale qui permet d'étendre les preuves par tableau aux logiques temporelles à temps linéaire est de séparer une formule en deux parties comme expliqué page 40: d'un côté les sous-formules à vérifier dans l'instant présent, et de l'autre (c'est-à-dire derrière l'opérateur X) celles à vérifier à l'instant suivant. Les opérateurs F, G, U et R peuvent être réécrits de façon à faire apparaître X grâce aux équations 3.1 page 40.

Les dernières lignes du tableau 4.1 présentent les règles de tableau pour les opérateurs LTL, construites à partir des équations susdites. (Les termes en P indiquent des *promesses* à tenir au sens de la page 41, nous pouvons les ignorer dans un premier temps.)

[4] Il s'agit de la formule $(A \Rightarrow B) \Rightarrow ((A \wedge C) \Rightarrow (B \wedge C))$ réécrite pour faire disparaître les $\Rightarrow$.

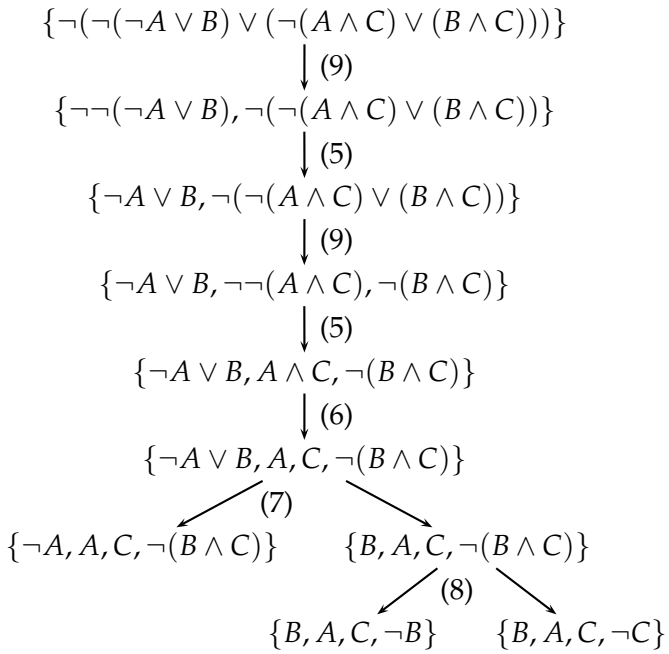


FIG. 4.6: Preuve de $\neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C))$ par tableau en appliquant successivement les règles (9), (5), (9), (5), (6), (7) et (8) du tableau 4.1.

Comme en logique propositionnelle, l'arbre est construit en appliquant les règles de tableau autant que possible, et en s'arrêtant sur des contradictions du type $\{p, \neg p\}$ ou $\{\bot\}$. Les quatre premiers nœuds de la figure 4.7 page ci-contre montrent un tel tableau. Lorsqu'on atteint un nœud irréductible, ne contenant que des variables propositionnelles atomiques ou leurs négations, ainsi que des formules commençant par X, on passe à l'*instant* suivant. C'est-à-dire que l'on construit un fils contenant toutes les formules qui étaient préfixées par X, mais sans ce X, et l'on déroule à nouveau l'algorithme à partir de ce nouveau nœud.

Ceci introduit de nombreuses différences entre la méthode des tableaux de la logique propositionnelle et celle pour LTL:
- Cette décomposition en *instant présent* et *instant suivant* introduit une partition du tableau en strates. Chaque strate correspond à un instant.
- Les équations (3.1) (page 40) étant récursives, le tableau correspondant devrait être un arbre infini. Le nombre d'étiquettes différentes étant par contre fini, on peut créer une représentation finie de cet arbre : un graphe dans lequel on identifie les nœuds identiques de l'arbre.
- Trouver une branche (potentiellement infinie) sans incohérence ne signifie pas que la formule f est satisfiable.

Ce dernier point découle du fait que les opérateurs temporels tels que F ou U introduisent des promesses que les réécritures découlant des équations (3.1) ne garantissent pas (cf. page 41). Par exemple le tableau va pouvoir boucler infiniment autour du nœud $\{f, \mathsf{X}(f\,\mathsf{U}\,g), \mathsf{P}\,g\}$ dans une branche qui ne satisfait jamais g. Ces cas seront détectés dans

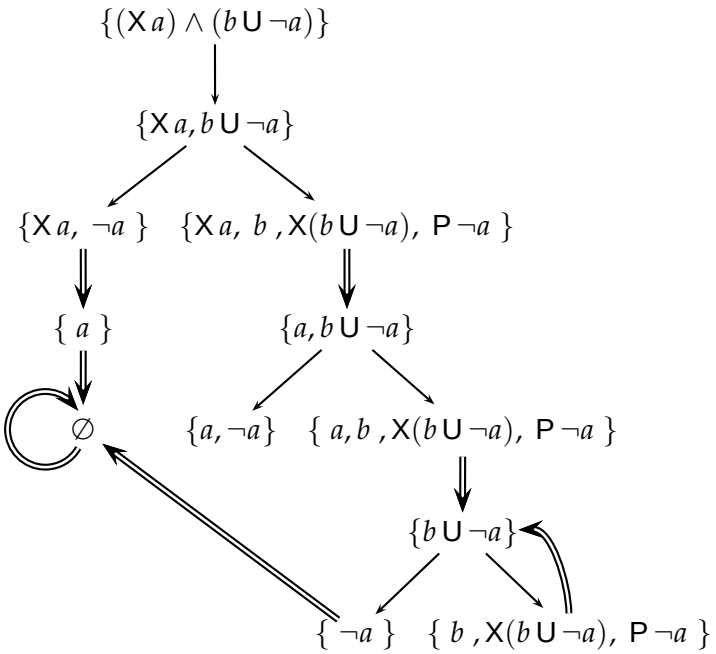


FIG. 4.7: Tableau pour $(\mathsf{X}\, a) \wedge (b\, \mathsf{U}\, \neg a)$. Les doubles flèches correspondent à un changement d'instant.

une seconde étape d'analyse effectuée après la construction du tableau, et c'est la raison pour laquelle nous avons explicitement introduit les promesses sous forme de termes dans les réécritures. Il faudra vérifier que chaque promesse est tenue. Un tableau est satisfait si toutes ses branches infinies tiennent leur promesses.

L'utilisation explicite de ces promesses dans la méthode des tableaux est originale [44]. Le mot promesse a été utilisé par Haddad et Vernadat [69] dans leur définition d'automates à promesses: des automates qui partagent la structure des TGBA mais dont les conditions d'acceptation, dites « promesses », sont duales, c'est-à-dire qu'un chemin acceptant ne doit pas promettre quelque chose continûment[5]. L'utilisation des promesses directement sous forme de termes dans les règles de tableau est inspirée de la traduction de Couvreur [34]. Traditionnellement, les méthodes de tableau pour LTL ont utilisé des moyens moins élégants et moins précis: pour se souvenir qu'un nœud correspond l'expansion d'une formule du type $f\, \mathsf{U}\, g$, on retient dans chaque nœud la liste des formules qui ont été développées [144, 145]. Le problème de conserver toutes formules réécrites dans les nœuds est que pour un instant donné les tableaux correspondants à des sous-formules différentes ne pourront par partager leurs feuilles. Le tableau résultant possède donc plus de nœuds. D'autres traductions ont ensuite cherché à réduire le nombre de formules à conserver dans les nœuds [61, 39], afin de favoriser leur fusion. Ces derniers algorithmes produisent des automates moins concis que ceux produits par les règles que nous donnons, mais la méthode est similaire.

[5] C'est exactement la même chose que les graphes instables de Michel [96] présentés section 4.1.2 page 68.

Construction d'un automate de Büchi généralisé étiqueté sur les transitions. La figure 4.8 page ci-contre montre comment le tableau construit pour une formule LTL peut être transformé en un automate de Büchi généralisé étiqueté sur les transitions. Rappelons que les doubles flèches représentent le passage à l'instant suivant et découpent le tableau en « strates ». Pour créer un automate de Büchi étiqueté sur les états, on considère les nœuds feuilles de chaque strate comme des états de l'automate, et on les connecte aux nœuds feuilles des strates suivantes. Les étiquettes des états sont les propositions atomiques que l'on trouve sur ces feuilles, et les conditions d'acceptation généralisées sont créées en complémentant les promesses.

Intuitivement, il est facile de comprendre la logique de cet algorithme. Une preuve par tableau montre la satisfiabilité d'une formule *par construction*. Chaque branche du tableau correspond à une séquence satisfaisant la formule. Dans le cas de la formule $(\mathsf{X}\, a) \wedge (b\, \mathsf{U}\, \neg a)$, il existe deux types d'exécutions acceptables:

- Si le premier état vérifie $\neg a$ et que le second état vérifie a, alors, quelle que soit la suite, l'exécution est acceptée.
- Enfin, si le premier état vérifie b, le second état vérifie $a \wedge b$ et qu'ils sont suivis d'un nombre *fini* d'états vérifiant b, puis d'un état vérifiant $\neg a$, alors l'exécution est acceptée quelle que soit la suite.

Ces deux types d'exécutions correspondent aux deux branches du tableau figure 4.8a, et se retrouvent donc dans l'automate de la figure 4.8b. Le caractère *fini* du nombre d'occurrences consécutives de b dans le préfixe du dernier type d'exécutions est imposé par les conditions d'acceptation de l'automate.

Un point mis en avant par Gerth et al. [61] est que cette construction d'automate peut se faire à la volée, par exemple pendant le calcul du produit synchronisé, au fur et à mesure des besoins de l'*emptiness check* (cf. page 28). Il suffit de conserver les informations caractérisant le nœud du tableau dans l'état correspondant de l'automate de Büchi. Lorsque les successeurs d'un état doivent être explorés, les règles de traduction sont appliquées localement sur les formules en X de l'état courant. De cette façon, il est possible de ne construire que la partie de l'automate qui est utilisée lors du produit synchronisé.

En pratique, le gain obtenu par la génération de l'automate *à la volée* mériterait une étude. Habituellement les formules à tester sont assez petites pour que la génération de l'automate complet ne pose pas de problème. Il est possible que le coût lié à la génération des états inutiles (ce travail n'est fait qu'une seule fois) ne dépasse pas celui dû à la gestion des états générés à la volée (faible, mais effectué à tout moment lors du produit synchronisé).

D'autre part, il faut rappeler que cet algorithme construit un automate de Büchi *généralisé*. Dans les approches traditionnelles, l'automate devrait d'abord être dégénéralisé (section 3.3.3 page 46) avant d'être synchronisé avec le système et enfin soumis à un algorithme d'*emptiness check*. L'opération de dégénéralisation peut elle aussi être effectuée à la volée, mais va multiplier les parcours de l'automate de Büchi d'origine.

Construction d'un automate de Büchi généralisé étiqueté sur les transitions. Il est naturellement possible de transformer en un TGBA l'automate généralisé étiqueté sur les états construit précédemment, en poussant les étiquettes des états sur les transitions sor-

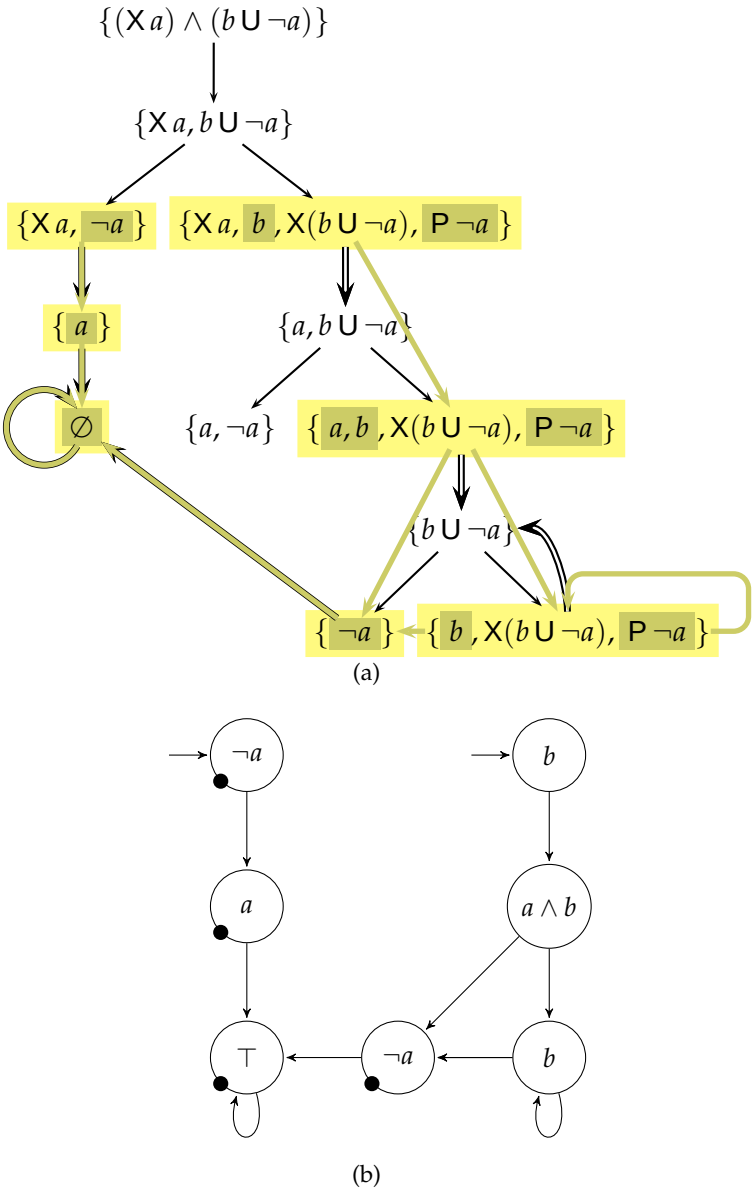


FIG. 4.8: Automate de Büchi étiqueté sur les états pour $(\mathsf{X}\, a) \wedge (b\, \mathsf{U}\, \neg a)$.

tantes (prop. 3 page 51). Une implémentation de cet algorithme est proposée par Rönkkö [103]. Il est cependant plus efficace d'interpréter le tableau comme illustré par la figure 4.9 page suivante.

Les états du TGBA correspondent aux racines de chaque strate du tableau, tandis que les propositions atomiques et promesses qui étiquettent les feuilles des strates servent à étiqueter les transitions reliant ces états. Sur l'automate de la figure 4.9b, nous avons reporté les formules LTL représentées par chaque état: ces formules ne font pas partie de la définition du TGBA mais elle en facilitent la lecture en indiquant simplement le langage reconnu à partir de cet état.

Ici encore, les ensembles de conditions d'acceptation sont générés en complémentant l'ensemble des transitions étiquetées par chaque promesse.

Cette traduction en TGBA correspond à l'algorithme de Couvreur [34, 36] à un détail près (sa réécriture de $f \mathsf{U} g$ promet $\mathsf{P}(f \mathsf{U} g)$ au lieu de $\mathsf{P}\, g$, donc manque d'occasionnelles simplifications).

4.1.4 Constructions par automates alternants

En 1988, Muller et Shupp [99] ont fait le lien entre la logique temporelle et une sous-classe d'ω-automates alternants: les automates alternants faibles.

Dans un automate de Büchi (non-déterministe) l'état suivant doit être choisi de façon non-déterministe parmi les successeurs. Ce choix peut être qualifié d'existentiel au sens où un mot sera accepté par l'automate s'il *existe* un successeur qui permet de reconnaître la suite du mot.

Les automates alternants combinent ce choix existentiel avec un choix universel dans lequel *tous* les successeurs doivent reconnaître la suite du mot. Dans son expression la plus simple, un automate alternant est composé d'états qui sont soit existentiels (un successeur, choisi de façon non-déterministe doit permettre d'accepter la suite du mot à reconnaître), soit universels (tous les successeurs doivent accepter la suite du mot). Le nom *alternant* vient de l'alternance entre ces choix existentiels et universels. Une façon d'interpréter l'universalité est d'imaginer que l'automate se clone pour chacun des successeurs d'une transition universelle pour continuer à lire la suite du mot dans chacun de ces états en parallèle. Le mot n'est accepté que si tous les clones créés l'acceptent. Un chemin dans un tel automate est généralement représenté par un arbre, avec un embranchement chaque fois qu'un choix universel force l'automate à se dupliquer.

Il existe une représentation plus concise qui homogénéise les choix existentiels et universels en autorisant les deux à cohabiter au sein du même état. Les transitions deviennent alors des hyper-transitions: elles possèdent chacune un seul état source, mais un ensemble de destinations. Le choix d'une hyper-transition parmi d'autres se fait de façon non-déterministe, puis la suite du mot doit être acceptée universellement par toutes ses destinations.

La figure 4.10a page 83 montre un exemple d'automate alternant très simple pour la formule $\mathsf{X}\, a \wedge (b \mathsf{U} \neg a)$. Les conventions que nous avons choisies pour représenter les automates alternants sont similaires à celles que nous utilisons pour les TGBA. Les hyper-

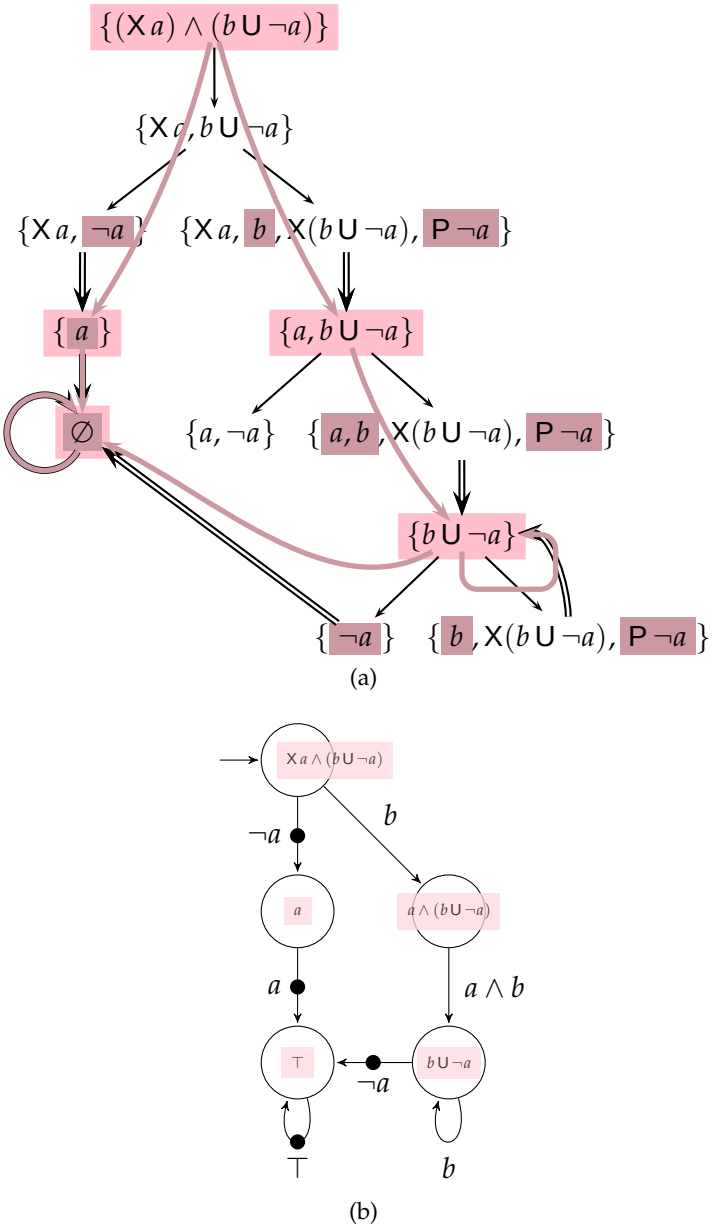


FIG. 4.9: Automate de Büchi étiqueté sur les transitions pour $(\mathsf{X}\, a) \wedge (b\, \mathsf{U}\, \neg a)$.

transitions sont représentées par des groupes de transitions accrochées au même point sur l'automate source. Ainsi si l'automate, dans son état initial q^0, lit une séquence qui commence par $\bar{a}b$, il devra choisir de façon non-déterministe entre le franchissement de la transition étiquetée par $\neg a$, ou celle étiquetée par b. S'il emprunte la première, il continuera la lecture de la séquence à partir de l'état q_1. S'il opte pour la seconde, il devra se dédoubler pour continuer la lecture à la fois depuis l'état q_1 et depuis l'état q_2. L'ensemble des destinations d'une hyper-transition peut aussi être réduit à l'ensemble vide, c'est le cas des deux transitions au bas de l'automate, et cela signifie qu'il n'y a plus besoin de vérifier quoi que ce soit ensuite.

La figure 4.10b montre un exemple d'arbre d'exécution acceptant n'importe quelle séquence débutant par $ab \cdot ab \cdot ab \cdot \bar{a}b \cdot \cdots$.

La formule $\mathsf{X}\,a \wedge (b\,\mathsf{U}\,\neg a)$, que nous avons traduite pour pouvoir faire ensuite le parallèle avec l'automate créé par la méthode des tableaux, est un cas particulier où tous les arbres d'exécution acceptants sont finis. Ceci est logique lorsqu'on regarde la formule: les sous-formules $\mathsf{X}\,a$ et $b\,\mathsf{U}\,\neg a$ doivent forcément être vérifiées après un nombre fini d'étapes, après quoi il n'y a plus rien à vérifier. Les états q_1 et q_2 correspondent à ces deux sous-formules, et les deux branches de l'exemple d'exécution disparaissent quand la formule correspondante a été vérifiée.

Dans le cas général, ces arbres d'exécution peuvent être infinis. Les figures 4.11a et 4.11b page 84 montrent un automate alternant pour la formule $\mathsf{F}\,\mathsf{G}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$ ainsi qu'une exécution de cet automate.

Les duplications de l'automate qui font suite aux franchissements d'hyper-transitions peuvent engendrer plusieurs copies de l'automate travaillant à partir du même état: il est inutile de conserver ces doublons car le résultat de chaque copie sera identique. L'arbre d'exécution peut donc être replié en un graphe acyclique en ne représentant les états qu'une fois à chaque étape. Une telle factorisation est visible au début de l'exécution de la figure 4.11b pendant la lecture de $\bar{a}\bar{b}$: l'état q_4 provient à la fois du franchissement d'une transition venant de q_2 et du franchissement de la boucle de q_4 sur lui-même.

Une exécution d'un automate alternant est acceptante si toutes ses branches infinies visitent chaque condition d'acceptation infiniment souvent. (Les exécutions finies telles que celle de la figure 4.11b page 84 sont trivialement acceptantes puisqu'elles n'ont pas de branche infinie.)

La traduction d'un tel automate alternant en TGBA peut se faire assez naturellement en faisant correspondre chaque état du TGBA à un ensemble d'états simultanément actifs de l'automate alternant. Les figures 4.10c et 4.11c montrent les TGBA correspondant à nos deux exemples.[6] Le calcul des transitions se fait en considérant toutes les combinaisons compatibles des transitions sortantes des états correspondants de l'automate alternant. Le TGBA construit avec cette construction peut avoir un nombre d'états exponentiellement plus grand que celui de l'automate alternant.

[6]Cette construction, due à Miyano et Hayashi [98], n'est pas sans rappeler l'algorithme classique de déterminisation d'un automate fini. La différence est que dans la déterminisation d'un automate fini, les ensembles d'états représentent une disjonction: les états parmi lesquels l'automate aurait dû faire un choix non-déterministe ; tandis que dans cette construction les états représentent des conjonctions. Autrement dit nous nous débarrassons ici des choix universels et non des choix existentiels liés à l'indéterminisme.

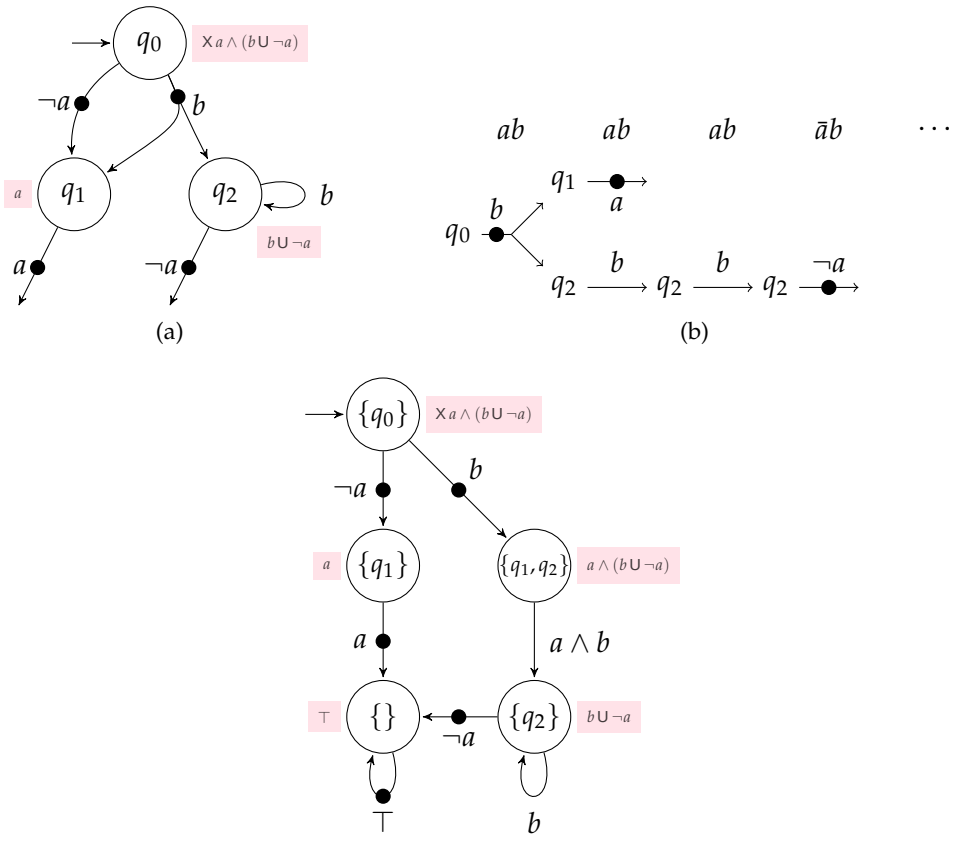


FIG. 4.10: (a) Automate alternant pour la formule $\mathsf{X}\,a \wedge (b \mathsf{U} \neg a)$; (b) un chemin de cet automate acceptant n'importe quelle séquence dont le préfixe est $ab \cdot ab \cdot ab \cdot \bar{a}b \cdots$; (c) TGBA généré à partir de l'automate alternant.

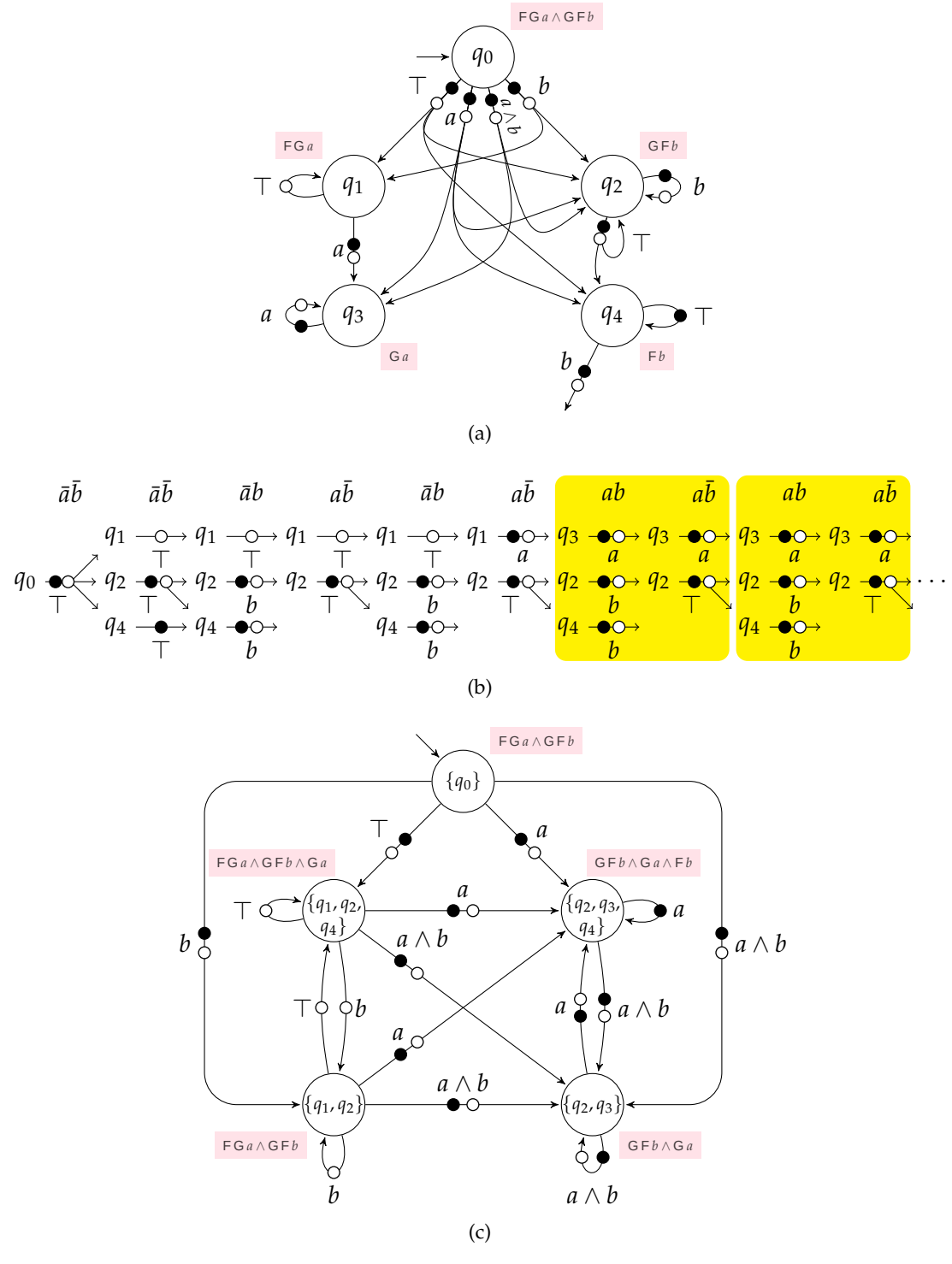


FIG. 4.11: (a) Automate alternant pour la formule $\mathsf{FG}\,a \wedge \mathsf{GF}\,b$; (b) un chemin de cet automate acceptant la séquence $\bar{a}\bar{b} \cdot \bar{a}\bar{b} \cdot \bar{a}b \cdot a\bar{b} \cdot \bar{a}b \cdot (a\bar{b} \cdot ab)^\omega$, le motif sur fond jaune s'y répète infiniment ; (c) TGBA construit à partir de l'automate alternant.

La construction de l'automate alternant correspondant à une formule LTL est moins évidente, mais elle peut se faire de façon compositionnelle. Pour chaque opérateur Tauriainen [123], section 3.1 montre de façon très claire comment créer un automate alternant à partir des automates alternants représentant ses opérandes. C'est cette traduction que nous avons utilisée pour construire les automates des figures 4.10a et 4.11a ; nous avons seulement inversé les conditions d'acceptation — qui sont utilisées comme des promesses dans sa méthode — pour rester plus proche des TGBA.

D'autres auteurs ont défini des constructions similaires, mais en utilisant des conditions d'acceptation sur les états [58, 100, 70, 54].

La taille de l'automate alternant est linéaire dans la taille de la formule traduite: comme le montrent les étiquettes que nous avons ajoutées à côté des états dans les figures, chaque état correspond à une sous-formule de la formule originale, et les hyper-transitions permettent d'exprimer n'importe quelle combinaison booléenne de ces sous-formules.

Les automates alternants résultant de la traduction d'une formule LTL ont la propriété suivante: les cycles de l'automate ne peuvent être que des boucles autour d'un état. Autrement dit, toutes les composantes fortement connexes de l'automate sont réduites à un état. Cette propriété s'explique intuitivement par le fait que les successeurs d'une formule ne peuvent être que des combinaisons de sous-formules, ou la formule elle-même, mais jamais une formule plus complexe. Cette classe d'automates alternants est dénommée différemment selon les auteurs: *very weak alternating automata* [58], *linear weak alternating automata* [70], *self-loop alternating automata* [123]... Elle est aussi expressive que LTL, et Tauriainen [123], section 3.4 donne un algorithme de traduction d'un tel automate en formule LTL.

Notons que les automates obtenus après traduction d'une formule LTL en automate alternant puis traduction d'un automate alternant en TGBA sont similaires à ceux obtenus par la traduction par tableau d'une formule LTL en TGBA. Le TGBA de la figure 4.10c page 83 ne diffère de celui de la figure 4.9 page 81 que par une condition d'acceptation sur une transition où elle n'a aucune influence sur les chemins acceptants. De même, le TGBA de la figure 4.11c page précédente est comparable à celui de la figure 4.12a page 89.

La traduction de la formule LTL en automate alternant fait apparaître de façon explicite les relations de récursivité entre les sous-formules (et les promesses associées). La traduction de l'automate alternant peut être vue comme la mise sous forme normale disjonctive. Dans la méthode de traduction par tableau ces deux opérations sont effectuées simultanément.

L'intérêt mis en avant par les partisans de l'approche par automates alternants est que cette étape intermédiaire permet d'effectuer des optimisations supplémentaires. Nous notons cependant que les simplifications données par Gastin et Oddoux [58], section 6 et Fritz [55], définition 1 pour supprimer des transitions inutiles peuvent être effectuées directement dans le tableau par un calcul d'impliquants premiers. La possibilité suggérée par Gastin et Oddoux [58], section 6 de réunir des états qui ont les mêmes successeurs, peut aussi être effectuée par l'approche tableau. Ces deux optimisations avaient déjà été proposées par Couvreur [34] dans sa traduction par tableau que nous détaillons section suivante.

Des algorithmes de réduction d'automates alternants basés sur des simulations ont été développés par Fritz et Wilke [56], mais ils expliquent plus tard [54] que cette réduction peut se faire directement pendant la traduction de l'automate alternant en automate de Büchi. Il semble naturel de se demander si une telle réduction ne serait pas possible dans les constructions par tableau.[7]

4.2 Traduction de J.-M. Couvreur

Nous avons déjà dit que la traduction d'un tableau en TGBA présentée section 4.1.3 page 78 correspond à l'algorithme de traduction introduit par Couvreur [34], et qu'utiliser $\mathsf{P}\,g$ au lieu de $\mathsf{P}(f\,\mathsf{U}\,g)$ dans la réécriture de l'opérateur U apportait une petite optimisation. Nous revoyons cet algorithme en détail afin de pouvoir y apporter d'autres modifications.

La représentation des nœuds d'un tableau proposée par Couvreur est basée sur les diagrammes de décision binaire (BDD pour *binary decision diagrams*) [22, 5]. Cette structure de données permet de représenter n'importe quelle expression booléenne de façon unique (c'est-à-dire que $a \rightarrow (b \wedge c)$ aura la même représentation que $(\neg a) \vee (b \rightarrow c)$). L'utilisation des BDD permet de simplifier la formule au niveau propositionnel (c'est-à-dire sans prendre en compte les opérateurs temporels), et aussi de la mettre facilement sous forme normale disjonctive.

Une formule LTL sera réécrite en un BDD où trois types de variables booléennes apparaissent:
– des variables du type $\mathrm{Var}[\varphi]$ représentant une proposition atomique φ,
– des variables du type $\mathrm{Prom}[\varphi]$ représentant $\mathsf{P}\,\varphi$ et
– des variables du type $\mathrm{Next}[\varphi]$ représentant $\mathsf{X}\,\varphi$.
Les règles de réécriture des opérateurs LTL (équation (4.2)) sont comparables aux règles de construction du tableau, présentées dans le tableau 4.1 page 75, dans le sens où elle vont transformer récursivement n'importe quelle formule LTL pour n'y faire apparaître que des propositions atomiques, des promesses, ou des formules LTL préfixées par X. Pour réduire leur nombre, nous supposons que la formule à traduire est sous forme normale positive (définition 16 page 39) et que les opérateurs F et G ont été remplacés par leurs définitions avec U et R.

[7] Nous ne sommes pas assez familier avec la théorie des jeux, sur laquelle repose le calcul des différentes relations d'équivalences basées sur les simulations, pour avancer dans cette voie.

$$
\begin{aligned}
B(\top) &= \top \\
eB(\bot) &= \bot \\
B(p) &= \mathrm{Var}[p] && \text{si } p \in AP \\
B(\neg p) &= \neg\mathrm{Var}[p] && \text{si } p \in AP \\
B(f \wedge g) &= B(f) \wedge B(g) \\
B(f \vee g) &= B(f) \vee B(g) \\
B(\mathsf{X}\, g) &= \mathrm{Next}[g] \\
B(f \,\mathsf{U}\, g) &= B(g) \vee (B(f) \wedge \mathrm{Next}[f \,\mathsf{U}\, g]) \wedge \mathrm{Prom}[g]) \\
B(f \,\mathsf{R}\, g) &= B(g) \wedge (B(f) \vee \mathrm{Next}[f \,\mathsf{R}\, g])
\end{aligned}
\tag{4.2}
$$

Exemple 18.

$$
\begin{aligned}
B\big(\mathsf{X}\, a \wedge (b \,\mathsf{U}\, \neg a)\big) &= B(\mathsf{X}\, a) \wedge B(b \,\mathsf{U}\, \neg a) \\
&= \mathrm{Next}(a) \wedge \big(B(\neg a) \vee (B(b) \wedge \mathrm{Next}[b \,\mathsf{U}\, \neg a] \wedge \mathrm{Prom}[\neg a])\big) \\
&= \mathrm{Next}[a] \wedge (\neg\mathrm{Var}[a] \vee (\mathrm{Var}[b] \wedge \mathrm{Next}[b \,\mathsf{U}\, \neg a] \wedge \mathrm{Prom}[\neg a]))
\end{aligned}
$$

En mettant l'expression sous forme normale disjonctive, on retrouve les termes qui correspondent aux deux feuilles de la première strate du tableau de la figure 4.8a page 79:

$$
= \underbrace{(\mathrm{Next}[a] \wedge \neg\mathrm{Var}[a])} \vee \underbrace{(\mathrm{Next}[a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[b \,\mathsf{U}\, \neg a] \wedge \mathrm{Prom}[\neg a])}
$$

Cette représentation symbolique (et unique) de la formule sert deux objectifs:

– D'une part elle permet de calculer les successeurs correspondant à un nœud étiqueté par une formule LTL. Comme le montre l'exemple 18, la mise sous forme normale disjonctive permet de retrouver l'équivalent des feuilles d'une strate du tableau.

  Pour limiter le nombre de successeurs, nous aurons intérêt à choisir une forme normale disjonctive sans redondance, c'est-à-dire où tous les impliquants sont premiers. Pour effectuer cette opération sur les BDD, nous pouvons utiliser par exemple les algorithmes de Minato [97] ou de Bouquet et al. [20].

– D'autre part, elle servira à étiqueter les nœuds du tableau. Cela procure un avantage sur la construction par tableau telle qu'elle a été décrite section 4.1.3 page 78: deux états correspondant à des formules différentes mais dont la représentation symbolique est la même seront identifiés. C'est naturel puisque cela signifie qu'ils expriment la même contrainte.

  Typiquement, des formules telles que $\mathsf{G}\,\mathsf{F}\,a$ et $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{F}\,a$ seront identifiées sans avoir besoin de savoir que la sémantique de LTL nous assure que $\mathsf{G}\,\phi \implies \phi$. L'exemple 19 ci-dessous déroule les calculs pour ces deux formules, et la figure 4.12 page 89 illustre la réduction apportée sur la formule $\mathsf{F}\,\mathsf{G}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$.

Exemple 19.

$$\begin{aligned} B(\mathsf{GF}\,a) &= B(\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)) \\ &= B(\top\,\mathsf{U}\,a) \wedge (\cancel{B(\bot)} \vee \mathrm{Next}[\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)]) \\ &= \underbrace{\underbrace{(B(a) \vee (\cancel{B(\top)} \wedge \mathrm{Next}[\top\,\mathsf{U}\,a] \wedge \mathrm{Prom}[a]))}_{\varphi_1} \wedge \mathrm{Next}[\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)]}_{\varphi_2} \end{aligned}$$

soit sous forme normale disjonctive:

$$= (\mathrm{Var}[a] \wedge \mathrm{Next}[\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)]) \vee (\mathrm{Next}[\top\,\mathsf{U}\,a] \wedge \mathrm{Prom}[a] \wedge \mathrm{Next}[\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)])$$

D'autre part

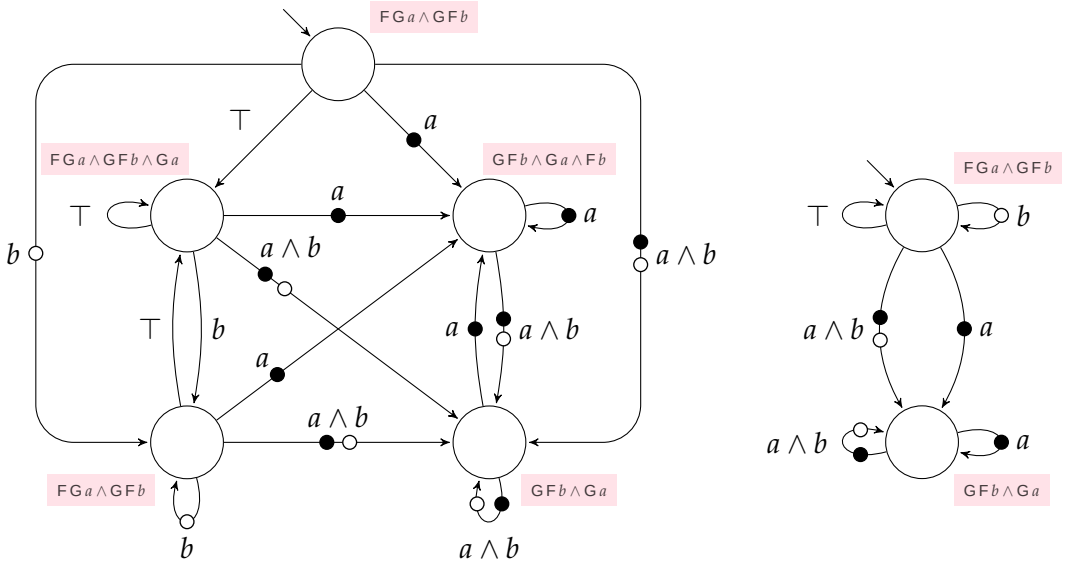
$$\begin{aligned} B(\mathsf{GF}\,a \wedge \mathsf{F}\,a) &= B((\bot\,\mathsf{R}(\top\,\mathsf{U}\,a)) \wedge (\top\,\mathsf{U}\,a)) \\ &= B((\bot\,\mathsf{R}(\top\,\mathsf{U}\,a))) \wedge B(\top\,\mathsf{U}\,a) \end{aligned}$$

opérandes déjà calculés

$$\begin{aligned} &= \varphi_2 \wedge \varphi 1 \\ &= \varphi_2 \end{aligned}$$

La figure 4.13 page 90 montre l'algorithme de traduction. Les états y sont représentés par les BDD provenant de leur réécriture par la fonction B de l'équation (4.2). L'ensemble *todo* liste des BDD représentant les états qui n'ont pas encore été traduits. Une fois qu'un BDD est retiré de l'ensemble (ligne 10), il est décomposé sous la forme de conjonction d'impliquants premiers. Chaque impliquant correspond à une transition et donne un ensemble V de propositions atomiques qui doivent être vérifiées, un ensemble V' de propositions atomiques qui ne doivent pas l'être, un ensemble A de promesses et enfin un ensemble N de formules à vérifier à l'instant suivant. Ceci nous autoriserait à définir une transition étiquetée par la conjonction $\bigwedge_{v\in V} v \wedge \bigwedge_{v\in V'} \neg v$. Cependant, comme notre définition des TGBA permet d'étiqueter chaque arc par une formule propositionnelle (pas seulement une conjonction) nous allons regrouper toutes les hypothétiques transitions qui partagent la même destination et les mêmes promesses. (La figure 2.5a page 23 donne un exemple d'automate où la transition étiquetée par $r_1 \vee \neg d_1$ provient en fait du regroupement de deux transitions étiquetées l'une par r_1 et l'autre par $\neg d_1$.) Nous procédons donc en deux étapes: la boucle des lignes 12–14 génère un ensemble de paires (conjonction à vérifier, variables Acc et Next), puis la boucle des lignes 15–22 parcourt cet ensemble en réunissant toutes les conjonctions des paires dont le second élément est identique pour enfin déclarer chaque transition.

Lors de la définition des transitions, ligne 21, A représente toujours un ensemble de promesses. Le véritable ensemble de conditions d'acceptation, $\mathcal{F} \setminus A$, y sera substitué ligne 23 une fois que l'ensemble $\mathcal{F}$ sera complètement déterminé.



(a) (b)

FIG. 4.12: Illustration de la réduction obtenue en fusionnant les états qui possèdent la même représentation symbolique. Ces états se repèrent sur la figures à ce qu'ils ont exactement les mêmes successeurs (états et étiquettes des transitions y menant). (a) Automate obtenu en appliquant la méthode des tableaux à la formule $\mathsf{FG}\,a \wedge \mathsf{GF}\,b$; (b) automate obtenu par l'algorithme de la figure 4.13 page suivante. Les trois états de gauche de l'automate (a) correspondent à l'état initial de l'automate (b), les deux autres états de (a) correspondent au deuxième état de (b). Notons que dans l'automate équivalent de la figure 4.11c page 84, seuls les deux états les plus à gauche possèdent les mêmes transitions sortantes, de petites variations sur la position des conditions d'acceptation empêchent l'identification des autres états.

1 **Entrées** : Une formule $f \in \mathrm{LTL}_{AP}$
2 **Résultat** : Un TGBA
3 **Données** : $todo \leftarrow \emptyset$
$\mathcal{Q} \leftarrow \emptyset$
$\mathcal{Q}^0 \leftarrow \emptyset$
$\mathcal{F} \leftarrow \emptyset$
$\delta \leftarrow \emptyset$
4 **begin**
5 $r \leftarrow B(f)$
6 $\mathcal{Q}^0 \leftarrow \{r\}$
7 $\mathcal{Q} \leftarrow \{r\}$
8 $todo \leftarrow \{r\}$
9 **while** $todo \neq \emptyset$ **do**
10 pick one src off $todo$
11 $succ \leftarrow \emptyset$
12 **foreach** $i \in prime\_implicants\_of(src)$ **do**
13 decompose i as $\bigwedge_{v\in V} \mathrm{Var}[v] \wedge \bigwedge_{v\in V'} \neg\mathrm{Var}[v] \wedge \bigwedge_{a\in A} \mathrm{Prom}[a] \wedge \bigwedge_{n\in N} \mathrm{Next}[n]$
14 $succ \leftarrow succ \cup \{(\bigwedge_{v\in V} v \wedge \bigwedge_{v\in V'} \neg v, \bigwedge_{a\in A} \mathrm{Prom}[a] \wedge \bigwedge_{n\in N} \mathrm{Next}[n])\}$
15 **foreach** d *such that* $\exists(p,d) \in succ$ **do**
16 decompose d as $\bigwedge_{a\in A} \mathrm{Prom}[a] \wedge \bigwedge_{n\in N} \mathrm{Next}[n]$
17 $dest \leftarrow B(\bigwedge_{n\in\mathbb{N}} n)$
18 **if** $dest \notin \mathcal{Q}$ **then**
19 $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{dest\}$
20 $todo \leftarrow todo \cup \{dest\}$
21 $\delta \leftarrow \delta \cup \left\{\left(src, \bigvee_{(p,d)\in succ} p, A, dest\right)\right\}$
22 $\mathcal{F} \leftarrow \mathcal{F} \cup A$
23 $\delta \leftarrow \{(s,p,\mathcal{F}\setminus a,d) \mid (s,p,a,d) \in \delta\}$
24 **return** $\langle 2^{AP}, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta\rangle$
25 **end**

FIG. 4.13: Algorithme de traduction d'une formule LTL en un TGBA [34]. La fonction de réécriture B, utilisée ligne 5 et 17, est définie par l'équation 4.2 page 87. Ligne 12, l'extraction d'implicants premiers à partir d'un BDD peut se faire par exemple avec la technique de Minato [97].

Thirioux [128] propose un algorithme de traduction appelé BOAM, assez proche de celui-ci en ce qu'il utilise aussi des BDD pour représenter les propositions qui étiquettent les arcs. Cependant ses états et la réécriture des formules qui les étiquettent ne sont pas représentés par des BDD, et les conditions d'acceptation sont ajoutées séparément en fonction des sous-formules qui étiquettent un état. Plusieurs des simplifications qu'il effectue après coup sur l'automate sont effectuées automatiquement dans la méthode ci-dessus par le simple fait de réécrire une formule LTL sous la forme de BDD.

4.3 Améliorations

Nous avons implémenté l'algorithme précédent dans Spot (annexe A page 201) avec plusieurs améliorations. Les sections suivantes détaillent ces améliorations isolément. L'influence relative de chacune sur le processus de vérification sera mesurée ensuite dans la section 4.4 page 98.

4.3.1 Amélioration du déterminisme

L'automate de la figure 3.7 page 50 n'est pas déterministe, et nous avons montré 3.11 page 57 qu'il existait un automate déterministe acceptant le même langage.

Du point de vue du processus de vérification, l'avantage d'un automate déterministe est qu'il limite l'explosion combinatoire lors du produit synchronisé. Par conséquent, l'algorithme d'*emptiness check* doit explorer moins d'états.

Comme expliqué section 3.3.6 page 56, la classe des automates de Büchi déterministes est moins expressive que celle des automates de Büchi. Il n'est pas toujours possible de produire un automate déterministe, mais nous pouvons essayer de le favoriser.

Soit $AP = \{a, b\}$ l'ensemble des propositions atomiques et $\Sigma = 2^{AP}$ l'alphabet. Supposons que l'on souhaite traduire la formule $a \,\mathsf{U}\, b$ en TGBA:

$$B(a \,\mathsf{U}\, b) = \mathrm{Var}[b] \vee (\mathrm{Var}[a] \wedge \mathrm{Next}[a \,\mathsf{U}\, b] \wedge \mathrm{Prom}[b]) \tag{4.3}$$

Si l'on considère cette formule sous forme normale disjonctive, l'état initial possède deux transitions sortantes: l'une étiquetée par b (la formule représentant $\{ab, \bar{a}b\}$) et à destination d'un état acceptant tout suffixe, et l'autre étiquetée par a (i.e., $\{ab, a\bar{b}\}$) bouclant sur l'état. La figure 4.14a page suivante représente cet automate.

Il est inutile de boucler sur l'état initial lorsque l'automate lit ab, car la formule $a \,\mathsf{U}\, b$ est alors vérifiée: n'importe quel suffixe peut être accepté. L'automate pourrait être simplifié comme indiqué dans la figure 4.14b. Cela correspond à la formule

$$\mathrm{Var}[b] \vee (\mathrm{Var}[a] \wedge (\neg\mathrm{Var}[b]) \wedge \mathrm{Next}[a \,\mathsf{U}\, b] \wedge \mathrm{Prom}[b]) \tag{4.4}$$

Cependant comme les équations (4.3) et (4.4) sont logiquement équivalentes, elles possèdent la même représentation sous forme de BDD. Si nous voulons produire l'automate de la figure 4.14b, c'est lors de l'extraction des différents impliquants que nous devons agir.

FIG. 4.14: Deux automates acceptant $a \mathsf{U} b$. L'automate (b) est déterministe ; (a) ne l'est pas.

L'idée est de considérer la formule à traduire pour chaque valuation des propositions atomiques (i.e., chaque lettre de Σ) possible. Au lieu de créer les transitions à partir de $B(a \mathsf{U} b)$, nous créons les transitions à partir de $B(a \wedge b \wedge (a \mathsf{U} b))$, $B(a \wedge (\neg b) \wedge (a \mathsf{U} b))$, $B((\neg a) \wedge b \wedge (a \mathsf{U} b))$ et $B((\neg a) \wedge (\neg b) \wedge (a \mathsf{U} b))$. Nous comptons ensuite sur le fait que certaines de ces transitions seront regroupées (ligne 21 de l'algorithme de la figure 4.13 page 90).

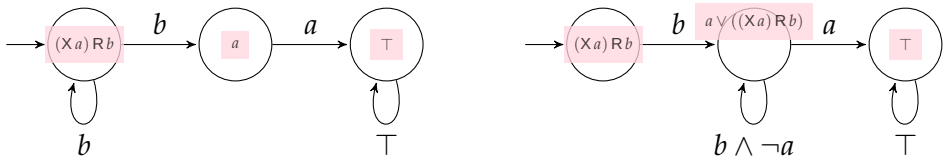
$$B(a \wedge b \wedge (a \mathsf{U} b)) = \mathrm{Var}[a] \wedge \mathrm{Var}[b] \tag{4.5}$$
$$B(a \wedge (\neg b) \wedge (a \mathsf{U} b)) = \mathrm{Var}[a] \wedge (\neg \mathrm{Var}[b]) \wedge \mathrm{Next}[a \mathsf{U} b] \wedge \mathrm{Prom}[b] \tag{4.6}$$
$$B((\neg a) \wedge b \wedge (a \mathsf{U} b)) = (\neg \mathrm{Var}[a]) \wedge \mathrm{Var}[b] \tag{4.7}$$
$$B((\neg a) \wedge (\neg b) \wedge (a \mathsf{U} b)) = \bot \tag{4.8}$$

Les équations (4.5) et (4.7), qui correspondent à des transitions possédant la même destination et les mêmes promesses, seront regroupées par la ligne 21.

Ces calculs, exponentiels dans le nombre de propositions atomiques, peuvent être accélérés de deux façons:

– Tout d'abord, il n'est nécessaire de traduire la formule LTL φ étiquetant l'état qu'une seule fois, car $B(\mathit{propositions} \wedge \varphi) = B(\mathit{propositions}) \wedge B(\varphi)$. Dans notre exemple, on calculera donc $\mathrm{Var}[a] \wedge \mathrm{Var}[b] \wedge B(a \mathsf{U} b)$, puis $\mathrm{Var}[a] \wedge (\neg \mathrm{Var}[b]) \wedge B(a \mathsf{U} b)$, etc.
– D'autre part, il n'est pas nécessaire d'énumérer toutes les valuations des propositions atomiques possibles: il suffit de ne considérer que celles qui peuvent être satisfaites. (Dans l'exemple nous voulons ignorer $\bar{a}\bar{b}$.) Une opération offerte par toute bibliothèque de BDD permet de trouver rapidement quelles sont les combinaisons de variables qui peuvent satisfaire une formule.

Une technique similaire a été utilisée par Sebastiani et Tonetta [108, 109] en travaillant directement sur le tableau représenté explicitement. Nous pensons que l'utilisation des BDD est beaucoup plus intéressante dans le sens où elle permet de se rendre compte rapidement qu'une formule est fausse, et qu'il est possible d'éviter de parcourir l'ensemble des 2^{AP} possibilités.



(a) $A_{(\mathsf{X}\,a)\,\mathsf{R}\,b}$ (b) $A_{b\wedge\mathsf{X}(a\,\mathsf{R}\,b)}$

FIG. 4.15: TGBA produits par les traductions de $(\mathsf{X}\,a)\,\mathsf{R}\,b$ et $b \wedge \mathsf{X}(a\,\mathsf{R}\,b)$, deux formules LTL équivalentes. Aucun de ces automates ne possède de conditions d'acceptation (tous les chemins infinis sont acceptants). Le second automate est déterministe, alors que le premier ne l'est pas.

4.3.2 Branching postponement

La figure 4.15 montre les automates représentant deux formules LTL équivalentes après traduction avec optimisation du déterminisme. L'automate de la figure 4.15a n'est pas déterministe: l'état initial possède deux transitions sortantes étiquetées par b. L'automate de la figure 4.15b en revanche est déterministe. Puisque les deux automates sont équivalents, on préférera donc utiliser le second.

La technique présentée dans cette section permet de construire le second automate pour l'une ou l'autre des formules. Il s'agit d'une technique présentée à l'origine par Sebastiani et Tonetta [108, 109] pour des automates de Büchi et que nous avons adaptée aux TGBA. Malheureusement, il ne s'agit que d'une heuristique: le déterminisme ne sera pas systématiquement amélioré, et l'automate produit sera parfois même plus gros.

L'idée est de regrouper les transitions sortantes d'un état qui sont étiquetées de la même façon (mêmes propriétés à vérifier, même promesses) en une transition allant vers un état dont la formule est la conjonction des formules étiquetant les destinations des formules d'origine.

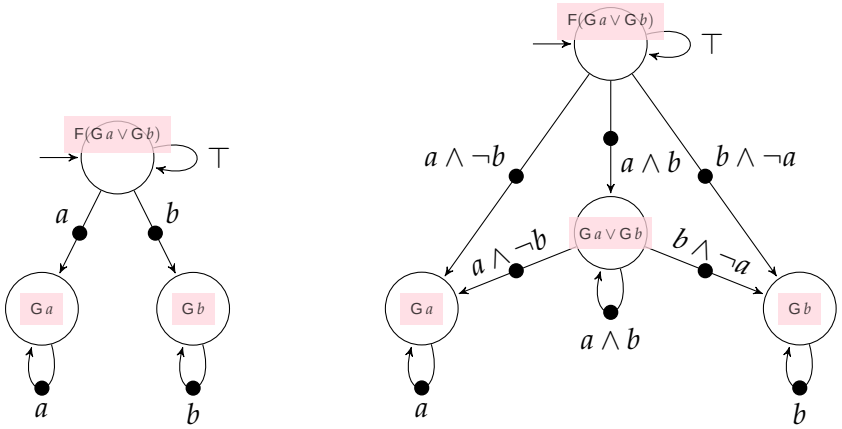
$$B\big((\mathsf{X}\,a)\,\mathsf{R}\,b\big) = \mathrm{Var}[b] \wedge \big(\mathrm{Next}[a] \vee \mathrm{Next}[(\mathsf{X}\,a)\,\mathsf{R}\,b]\big)$$

Ici nous avons deux transitions étiquetées par b, l'une allant vers un état devant vérifier a, l'autre vers un état devant vérifier $(\mathsf{X}\,a)\,\mathsf{R}\,b$. Cela donne les deux transitions sortantes de l'état initial de la figure 4.15a. Si nous appliquons le *branching postponement*, ces deux variables Next sont réécrites en une seule:

$$= \mathrm{Var}[b] \wedge \big(\mathrm{Next}[a \vee (\mathsf{X}\,a)\,\mathsf{R}\,b]\big)$$

Cette réécriture donne la transition entre les deux premiers états de l'automate de la figure 4.15a. Les autres transitions proviennent des réécritures et de l'application de la technique de la section 4.3.1 page 91 pour favoriser le déterminisme.

$$\begin{aligned} B\big(a \vee (\mathsf{X}\,a)\,\mathsf{R}\,b\big) &= \mathrm{Var}[a] \vee \big(\mathrm{Var}[b] \wedge \big(\mathrm{Next}[a] \vee \mathrm{Next}[(\mathsf{X}\,a)\,\mathsf{R}\,b]\big)\big) \\ &= \mathrm{Var}[a] \vee \big(\mathrm{Var}[b] \wedge \big(\mathrm{Next}[a \vee (\mathsf{X}\,a)\,\mathsf{R}\,b]\big)\big) \end{aligned}$$



(a) (b)

FIG. 4.16: TGBA pour la formule $\mathsf{F}(\mathsf{G}\,a \vee \mathsf{G}\,b)$: (a) sans *branching postponement*, (b) avec.

Une formule qui souffre de cette heuristique est $\mathsf{F}(\mathsf{G}\,a \vee \mathsf{G}\,b)$. La figure 4.16 montre les automates la traduisant, avec et sans *branching postponement*. La combinaison du *branching postponement* avec la considération des différentes combinaisons de variables propositionnelles possibles pour favoriser le déterminisme engendre ici un nouvel état.

4.3.3 Simplifications de formules

Plusieurs auteurs ont proposé des règles de simplification de formules LTL. Ces règles s'appliquent avant la traduction et cherchent à réduire la formule dans l'espoir de simplifier le travail de traduction.

Somenzi et Bloem [112], Etessami et Holzmann [49] et Bloem [18] proposent des systèmes de réécriture de la formule qui permettent, de façon purement syntaxique, une réduction basée sur les propriétés des opérateurs de logique temporelle et une réduction au niveau propositionnel. Les réécritures proposées par Somenzi et Bloem d'une part, et Etessami et Holzmann d'autre part ne se recoupent pas totalement.

La plupart de ces règles s'appuient sur une détection syntaxique de l'implication entre deux sous-formules. Par exemple la formule $\varphi\,\mathsf{U}\,\psi$ peut être réécrite simplement ψ si l'on peut établir que $\varphi \implies \psi$.

Plutôt que de détecter cette implication syntaxiquement (les algorithmes proposés ne couvrent pas tous les cas), Tauriainen [123], section 5.3 propose d'utiliser l'approche automate pour tester l'inclusion des langages correspondant à la formule LTL. Tester si $\varphi \implies \psi$ revient à tester si $\mathscr{L}(\varphi) \subseteq \mathscr{L}(\psi)$, c'est-à-dire à tester si $\mathscr{L}(A_\varphi \otimes A_{\neg\psi}) = \emptyset$.

Les règles proposées par Tauriainen ne parviennent cependant pas à simplifier des formules comme $\mathsf{F}\,\mathsf{F}\,\varphi$, $\mathsf{G}\,\mathsf{G}\,\varphi$, $\mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\mathsf{G}\,\varphi$, $\mathsf{G}(\varphi\,\mathsf{R}\,\psi)$, ou $\mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\mathsf{X}\,\varphi$ équivalentes respectivement à $\mathsf{F}\,\varphi$, $\mathsf{G}\,\varphi$, $\mathsf{F}\,\mathsf{G}\,\varphi$, $\mathsf{G}\,\psi$ et $\mathsf{G}\,\mathsf{F}\,\varphi$. Les règles de simplifications que nous proposons dans le tableau 4.2 page suivante améliorent celles de Tauriainen dans les cas cités.

motif	conditions à tester	simplification
$\varphi \vee \psi$	$\mathscr{L}(A_\varphi \otimes A_{\neg\psi}) = \emptyset$	ψ
	$\mathscr{L}(A_{\neg\varphi} \otimes A_\psi) = \emptyset$	φ
	$\mathscr{L}(A_{\neg\varphi} \otimes A_{\neg\psi}) = \emptyset$	$\top$
$\varphi \wedge \psi$	$\mathscr{L}(A_\varphi \otimes A_{\neg\psi}) = \emptyset$	φ
	$\mathscr{L}(A_{\neg\varphi} \otimes A_\psi) = \emptyset$	ψ
	$\mathscr{L}(A_\varphi \otimes A_\psi) = \emptyset$	$\top$
$\varphi \,\mathsf{U}\, \psi$	$\mathscr{L}(A_{\varphi \mathsf{U} \psi} \otimes A_{\neg\psi}) = \emptyset$	ψ
	$\mathscr{L}(A_{\neg\varphi} \otimes A_{\neg\psi}) = \emptyset$	$\mathsf{F}\,\psi$
$\varphi \,\mathsf{R}\, \psi$	$\mathscr{L}(A_{\neg(\varphi \mathsf{R} \psi)} \otimes A_\psi) = \emptyset$	ψ
	$\mathscr{L}(A_\varphi \otimes A_\psi) = \emptyset$	$\mathsf{G}\,\psi$
$\mathsf{X}\,\varphi$	$\mathscr{L}(A_\varphi \otimes A_{\neg \mathsf{X}\varphi}) = \emptyset \wedge \mathscr{L}(A_{\neg\varphi} \otimes A_{\mathsf{X}\varphi}) = \emptyset$	φ

TAB. 4.2: Règles de simplification de formules LTL.

Ces règles sont appliquées sur l'arbre syntaxique représentant la formule LTL à partir de ses feuilles et en remontant vers la racine. Les automates construits pour chaque sous-formule sont conservés dans un cache afin de pouvoir être réutilisés. Ce cache est d'ailleurs la raison pour laquelle on teste si $\mathscr{L}(A_\varphi \otimes A_{\neg\psi}) = \emptyset$ au lieu de tester directement si $\mathscr{L}(A_{\neg(\varphi\to\psi)}) = \emptyset$: les automates construits pour les sous-formules servent plusieurs fois.

La bibliothèque Spot (annexe A page 201) implémente ces règles simplificatrices ainsi que celles de la littérature. Dans un premier temps, nous pouvons juger de leur effet au niveau de formules LTL en les appliquant sur un ensemble de formules LTL et en mesurant la réduction apportée selon la formule $1 - \frac{|simpl(\varphi)|}{|\varphi|}$. La tableau 4.3 page suivante donne la moyenne des réductions apportées par différentes combinaisons de simplifications, sur deux jeux de formules LTL différentes. Les formules aléatoires sont un ensemble de 1200 formules LTL générées aléatoirement avec des tailles comprises entre 10 et 20. Les formules prédéfinies correspondent à un ensemble de 94 formules LTL tirées de la littérature [45, 49, 112].

L'algorithme de génération aléatoire d'une formule LTL de taille n (comptée en nombre d'opérateurs logiques et de propriétés atomiques) travaille récursivement. Une formule de taille 1 est une proposition atomique tirée au hasard. Si $n > 1$ l'algorithme tire un opérateur (booléen ou temporel) au hasard (en se limitant aux opérateurs unaires si $n = 2$) puis construit les opérandes de façon à ce que leurs tailles combinées fassent $n - 1$ (par exemple pour un opérateur binaire on tire une valeur aléatoire m entre 1 et $n - 2$, puis on génère une formule de taille m pour l'opérande gauche puis une autre de taille $n - 1 - m$ pour l'opérande droite). Les probabilités respectives de chaque opérateur et des propositions atomiques sont configurables, mais dans ces tests nous les gardons équiprobables.

Les numéros des règles de réduction correspondent aux simplifications suivantes:

	formules	
règles de réduction	aléatoires	prédéfinies
(1)	15.27%	0.83%
(2)	44.47%	5.09%
(3)	5.34%	0.62%
(1)+(2)+(3)	61.59%	7.63%
(4)	59.60%	9.08%
(5)	62.95%	9.70%
(1)+(2)+(3)+(5)	63.66%	10.18%
(1)+(5)	63.82%	10.18%
(2)+(5)	62.79%	9.70%
(1)+(2)+(5)	63.66%	10.18%
(3)+(5)	62.95%	9.70%
(1)+(3)+(5)	63.82%	10.18%

TAB. 4.3: Évaluation de différentes règles de simplification de formules.

(1) Les réécritures de Somenzi et Bloem [112], qui ne demandent aucun test d'implication. Ce sont des réécritures fonctionnant uniquement par substitution de motifs. Par exemple $\mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\varphi$ peut toujours être remplacé par $\mathsf{G}\,\mathsf{F}\,\varphi$.

(2) Les réécritures de Somenzi et Bloem [112] basées sur des tests d'implication syntaxiques. Déjà citées, ces réécritures sont conditionnelles: par exemple le motif $\varphi\,\mathsf{U}\,\psi$ peut être réécrit en ψ si l'on peut prouver syntaxiquement que $\varphi \implies \psi$.

(3) Les simplifications des formules *purement eventuelles* ou *purement universelles* d' Etessami et Holzmann [49]. Ces termes correspondent à deux sous-classes de formules LTL caractérisables syntaxiquement et sur lesquelles certaines réductions peuvent être définies. Principalement $\varphi\,\mathsf{U}\,\psi \equiv \psi$ si ψ est purement éventuelle et $\varphi\,\mathsf{R}\,\psi \equiv \psi$ si ψ est purement universelle.

(4) Les règles de simplifications de Tauriainen [123], section 5.3, basées sur les tests d'inclusion de langage à l'aide d'automates.

(5) Nos règles du tableau 4.2 page précédente.

Le fait que le jeu de formules aléatoires se réduit beaucoup plus facilement que celui des formules prédéfinies n'est pas une surprise: ce dernier contient des formules écrites à la main pour des cas concrets, elles sont donc naturellement peu redondantes.

De façon générale, on peut noter que les règles (5) simplifient plus que les quatre autres individuellement, et que la combinaison (1)+(5) suffit à obtenir les réductions maximales (relativement aux autres).

D'un point de vue théorique, les réductions (4) et (5) sont beaucoup plus complexes que les trois autres parce que le test d'inclusion de langage qui y est fait demande une traduction des sous-formules LTL en automates de Büchi et que la taille de cette traduction est exponentielle dans la taille de la formule. En pratique, cette complexité n'est pas un problème car la taille des formules est modérée. Il est très rare que la réduction ne soit pas instantanée, et quand bien même l'algorithme y passerait quelques secondes cela est

très raisonnable si cela permet de réduire la taille de l'automate qui sera synchronisé avec l'espace d'état.

Simplification lors de la traduction. Nous avons jusqu'à présent considéré la simplification d'une formule LTL comme une étape antérieure à sa traduction en TGBA. Nous pouvons aussi envisager une simplification lors de la traduction.

Comme chaque état T de l'automate correspond à une formule LTL dont le langage est $\mathscr{L}(A[\{T\}])$, il semble naturel d'essayer de tirer parti de ces formules LTL pour effectuer des simplifications.

Bloem [18], section 3.4.3 donne un exemple montrant que si deux états acceptent le même langage, il est généralement faux de supprimer l'un des deux états (en redirigeant ses transitions entrantes vers l'autre). Ceci n'est correct que si l'état que l'on souhaite retirer n'est pas accessible depuis son remplaçant.

Une opération que nous pouvons faire sans danger en revanche, est de simplifier la formule LTL qui correspond à un état avant de s'en servir pour calculer les successeurs. Cela revient à simplifier la formule avant la décomposition de la ligne 12 dans l'algorithme 4.13 page 90.

Nos expérimentations montrent que ce dernier changement n'apporte aucune réduction que la simplification de la formule avant la construction n'ait pas déjà apportée. Par la suite nous ne considérerons donc que les simplifications effectuées avant la traduction.

4.3.4 Étude des composantes fortement connexes

Comme nous l'avons signalé précédemment (proposition 2 page 51), les conditions d'acceptation qui n'apparaissent dans aucun cycle de l'automate n'ont pas d'influence sur son langage et peuvent être retirées. En déterminant les composantes fortement connexes de l'automate (avec un algorithme proche de celui présenté section 5.4.1 page 120) nous pourrions ainsi supprimer des conditions d'acceptation superflues.

Nous n'avons pas cherché à mettre en pratique cette optimisation car elle ne réduit pas la taille du produit $\mathcal{A}_M \otimes \mathcal{A}_{\neg\varphi}$, elle ne peut que réduire le nombre de conditions d'acceptation utilisées. Or l'algorithme d'*emptiness check* que nous présenterons section 5.4.1 et que nous modifierons dans les chapitres suivants est insensible aux nombre de conditions d'acceptation utilisées.

Retirer des conditions d'acceptation peut cependant être bénéfique à d'autres algorithmes d'*emptiness check*. Ces algorithmes, que nous appellerons NDFS dans la chapitre suivant, sont basés sur plusieurs parcours en profondeurs imbriqués et déclenchés à partir de chaque transition étiquetée par une condition d'acceptation. Ainsi supprimer la condition d'acceptation « ● » qui étiquette inutilement la transition centrale des automates de la figure 4.14 page 92 améliorerait légèrement le temps d'exécution de ces *emptiness check* en évitant de démarrer un parcours en profondeur.

Une décomposition des ces composantes permet aussi de détecter (puis supprimer) des portions de l'automate à partir desquelles il est impossible de construire un chemin acceptant. Nous ne nous attardons pas sur la façon de déterminer ces ensembles d'états in-

utiles, car il s'agit essentiellement du même travail que celui des algorithmes d'*emptiness check*.

4.4 Comparatif

Le tableau 4.4 page suivante montre comment les traductions implémentées dans Spot se comparent aux autres algorithmes disponibles et mesure l'influence des différentes optimisations.

Pour faire ces mesures, nous avons utilisé LBTT [117, 118, 125], un outil permettant de tester les algorithmes de traduction de formules LTL en automates de Büchi. LBTT génère des formules LTL de façon aléatoire et récupère la traduction pour chaque algorithme, en comparant ces automates entre eux, et en les utilisant pour effectuer le *model checking* d'espaces d'état générés aléatoirement. LBTT peut ainsi détecter des erreurs de traduction. L'auteur de l'outil LBTT a ainsi découvert [124] une erreur dans le traducteur utilisé par Spin (basé sur l'algorithme d' Etessami et Holzmann [49]). LBTT nous a aussi été d'une aide précieuse lors du développement des algorithmes de traduction de Spot, et, en dehors de nos propres erreurs, il nous a montré des erreurs dans les implémentations de LTL2BA [58] et Modella [109] lors de nos tests.

Comme LBTT affiche les tailles des automates produits par chacun des algorithmes et pour chacune des formules LTL, ainsi que la taille du produit synchronisé de cet automate avec un espace d'état aléatoire (mais identique pour tous les traducteurs), il peut être détourné de son véritable usage pour produire les mesures du tableau 4.4.

Les colonnes sont groupées par rapport à trois jeux de données:

1. 100 formules LTL aléatoires de taille 10 avec 4 propositions atomiques,
2. 100 formules LTL aléatoires de taille 13–18 avec 8 propositions atomiques,
3. 94 formules LTL tirées de la littérature (celles que nous avons déjà utilisées section 4.3.3 page 94)

Pour chacun de ces jeux de données, nous indiquons plusieurs valeurs:

1. la somme des états des automates correspondant à toutes les formules traduites
2. la somme des transitions des automates correspondant à toutes les formules traduites
3. la somme des états des automates correspondant au produit de l'automate de la formule avec un espace d'état
4. la somme des transitions des automates correspondant au produit de l'automate de la formule avec un espace d'état
5. le cumul des secondes passées à traduire toutes les formules

La taille du produit (colonnes 3 et 4) nous permet de mieux évaluer l'influence des optimisations qui visent à améliorer le déterminisme.

Verticalement, nous listons diverses implémentations étrangères d'algorithmes de traduction, ainsi que celles de Spot avec diverses options d'optimisation. Certains de ces outils ne produisent que des automates dégénéralisés ; nous avons donc coupé la table

	Petites formules					Moyennes formules					Formules prédéfinies				
	Automate		Produit			Automate		Produit			Automate		Produit		
Traducteur	ét.	tr.	ét.	tr.	Tps	ét.	tr.	ét.	tr.	Tps	ét.	tr.	ét.	tr.	Tps
Traductions en automates de Büchi dégénéralisés:															
Spin	723	2134	143191	7761k	1.1	1954	20453	384554	40802k	474.4	>848	4197	166977	12265k	7234.4
Wring	820	1453	163925	6763k	10.7	1818	4575	362493	16203k	48.6	1180	8584	234973	24006k	59.8
LTL2BA	547	1172	109400	5148k	0.3	1141	4087	228158	12782k	0.3	644	2519	128593	8368k	0.3
LTL2NBA	**499**	991	99800	4393k	5.7	**1054**	3372	209977	10843k	16.4	543	2130	108384	7827k	537.0
Modella	574	1228	114695	4196k	0.2	1487	5935	294861	11809k	17.7	906	3320	179107	7514k	97.6
Spot/FM	641	1386	128197	5627k	2.8	1388	4695	277559	13904k	3.0	680	2395	134974	8655k	2.6
+(L)	603	1220	120597	4918k	3.4	1330	4493	265954	13143k	3.9	647	2272	128375	8121k	3.8
+(123)	596	1204	119197	4864k	2.7	1310	4450	261951	12841k	3.2	667	2366	132375	8334k	3.0
+(15)	593	1197	118597	4840k	3.3	1292	4382	258351	12652k	5.0	641	2244	127175	8064k	5.2
+(15)+(D)	593	1197	118478	4274k	3.2	1292	4382	256574	9954k	5.2	641	2244	126548	4893k	5.4
+(15)+(D)+(B)	596	1221	119078	4238k	3.5	1262	4226	250288	8944k	5.3	661	2362	130087	4992k	5.3
+(15)+(D)+(B)+(S)	525	1076	104909	3811k	5.3	1113	3739	220934	8152k	100.3	549	1810	107988	4324k	15.8
+(15)+(D) +(S)	**504**	1023	100736	3624k	5.9	1115	3633	221728	8786k	157.4	**531**	1712	104596	4212k	14.7
+(15) +(S)	505	1023	100996	4134k	6.3	**1110**	3590	221958	10813k	191.8	**527**	1639	104387	6274k	23.4
Traductions en automates de Büchi généralisés:															
LBT	1611	4498	321998	17172k	0.4	désactivé					3574	4816	704605	96066k	0.8
Wring	798	1427	159525	6619k	10.9	1642	4294	327546	14847k	48.6	792	3356	157836	11425k	59.8
LTL2BA	507	1014	101400	4510k	4.3	1006	3309	201178	10564k	4.7	496	1764	98997	6151k	4.2
Spot/LaCIM	962	5010	192400	10589k	2.7	3238	45388	647600	38493k	4.4	1167	9065	233200	13810k	3.1
+(15)	789	3739	157800	8483k	3.4	2490	31120	498000	29646k	5.7	967	7308	193200	11458k	5.4
Spot/FM	524	1086	104798	4422k	2.8	1024	3340	204785	9972k	3.2	468	1607	93395	5779k	3.0
+(L)	494	975	98798	3936k	3.4	977	3173	195376	9347k	4.0	444	1512	88596	5369k	3.8
+(123)	487	959	97398	3882k	2.6	963	3142	192574	9110k	3.2	466	1635	92996	5728k	3.0
+(15)	484	952	96798	3857k	3.4	952	3097	190374	8978k	5.0	446	1534	88996	5511k	5.1
+(15)+(D)	484	952	96707	3433k	3.3	952	3097	189347	7162k	5.0	446	1534	88569	3467k	5.4
+(15)+(D)+(B)	489	980	97707	3419k	3.2	954	3153	189444	6697k	5.2	466	1647	92183	3564k	5.3
+(15)+(D)+(B)+(S)	484	953	96712	3370k	3.8	945	3046	187634	6628k	11.0	436	1388	86373	3251k	6.7
+(15)+(D) +(S)	**478**	920	95568	3373k	3.8	**916**	2804	182140	6760k	65.2	**413**	1284	81999	3140k	9.9
+(15) +(S)	**479**	923	95798	3783k	3.8	**914**	2768	182774	8229k	73.9	**413**	1267	82397	4774k	13.8

TAB. 4.4: Évaluation des traducteurs de formules LTL.

en deux afin de marquer cette différence, et pour nous comparer facilement nous avons demandé à Spot de dégénéraliser ses automates.

Les nombres en gras repèrent les deux meilleures tailles pour chaque jeux de données, dans les cas généralisés ou non.

Les différents traducteurs évalués sont les suivants:

Spin 4.0.6 [75]. L'algorithme de traduction utilisé est celui de Gerth et al. [61] amélioré par Etessami et Holzmann [49]. À l'heure où nous écrivons ces lignes, la dernière version de Spin est 4.2.9 mais la liste des changements n'indique rien qui affecte la traduction.

Nous avons noté « >848 » pour le nombre d'états de la colonne des formules prédéfinies car Spin n'a pas pu traduire deux des formules fournies. Dans un cas le processus a été tué par le système après plus de 6h de calcul ; dans l'autre cas c'est nous qui l'avons tué après presque 3h d'attente.[8] Les sommes indiquées pour cette entrée portent donc sur 90 formules et non 92 comme les autres algorithmes. Le nombre d'états (et les autres mesures) serait donc supérieur si Spin avait réussit à traduire ces formules.

La formule est simplifiée par les règles appelées (3) dans la section 4.3.3 page 94. L'automate produit n'est pas simplifié ensuite.

Il s'agit d'un exécutable, appelé directement.

Wring 1.1.0 [112]. La formule est simplifiée avec les règles (1) et (2) de la section 4.3.3 page 94 avant d'être traduite à l'aide d'un tableau. L'automate est ensuite simplifié par simulation [18]. Enfin, l'automate peut-être dégénéralisé quand cela est demandé: nous avons donc mesuré les deux cas.

Cet outil, écrit en Perl, est donc interprété.

LTL2BA 1.0 [58]. L'analyse et la simplification de la formule ont été empruntées à Spin, et deux règles ont été ajoutées:

$$\varphi \wedge (\psi \,\mathsf{U}\, \varphi) \equiv \varphi$$
$$\varphi \vee (\psi \,\mathsf{R}\, \varphi) \equiv \varphi$$

La traduction produit ensuite un automate alternant (avec condition d'acceptation sur les états) qui est ensuite traduit en automate de Büchi généralisé (avec condition d'acceptation sur les transitions) et enfin dégénéralisé.

Les automates alternants et les automates de Büchi généralisés sont simplifiés en supprimant les transitions redondantes (transitions impliquées par d'autres) et en calculant les composantes fortement connexes pour supprimer conditions d'acceptation et états inutiles.

Ici encore, nous avons mesuré cet outil deux fois, pour les cas généralisé ou non.

Il s'agit d'un exécutable, appelé directement.

LBT 1.2.1 [103]. LBT implémente l'algorithme de Gerth et al. [61] sans aucune optimisation. Comme le suggèrent les résultats c'est probablement ce qui se fait de pire en

[8] Par acquis de conscience nous avons vérifié que le problème existait toujours avec la version 4.2.9.

matière de traduction LTL, mais il a l'avantage d'être très simple et peu donc servir de référence lorsqu'LBTT[9] est utilisé pour vérifier d'autres traducteurs.

Nous l'avons désactivé lors de la génération des moyennes formules pour gagner un peu de temps. Même si la traduction est très rapide, LBTT prend ensuite beaucoup de temps à vérifier les énormes automates produits. Au reste, les deux autres tests suffisent à montrer son infériorité...

Il s'agit d'un exécutable, appelé directement.

LTL2NBA [54]. Comme LTL2BA, LTL2NBA traduit la formule LTL en un automate alternant étiqueté sur les états. Cet automate est simplifié lors de sa conversion en automate de Büchi avec une relation de simulation « retardée » [56, 50] (qui laisse plus de liberté à la position des conditions d'acceptation).

Les formules sont préalablement simplifiées, mais nous ignorons par quelles règles.

Cet outil, écrit en Python, est donc interprété.

Modella 1.5.1 [109, 108] Modella est une construction par tableau sans simplification de formule préalable. La construction vise à favoriser le déterminisme de l'automate produit. L'automate dégénéralisé est simplifié avec les techniques de simulation d' Etessami et al. [50].

Il s'agit d'un exécutable, appelé directement.

Spot/LaCIM est notre implémentation de l'algorithme présenté par Couvreur [35] et que nous avons mentionné section 4.1.1 page 65.

Il s'agit d'un exécutable appelé au travers d'un script shell.

Spot/FM est notre implémentation de l'algorithme de la section 4.2 page 86.

Il s'agit d'un exécutable appelé au travers d'un script shell.

Pour ces deux derniers algorithmes nous avons détaillé différentes combinaisons d'optimisations, repérées par les clefs suivantes:

(123) Simplification de formules avec les règles (1)+(2)+(3) de la section 4.3.3 page 94.

(15) Simplification de formules avec les règles (1)+(5) de la section 4.3.3.

(L) Simplification de formules avec les mêmes règles que LTL2BA.

(D) Optimisation du déterminisme (section 4.3.1 page 91).

(B) *Branching postponement* (section 4.3.2 page 93).

(S) Réduction de l'automate en fonction de relations de simulations calculées selon les algorithmes d'Etessami et al. [50]. Cette réduction n'est malheureusement effectuée que sur les TGBA dont le nombre de conditions d'acceptation est inférieur ou égal à 1 et n'est pas aussi complète que l'orignale. Elle a été implémentée (ainsi que les simplifications de formules (1), (2) et (3)) par Thomas Martinez à l'occasion de son stage de DEA.

Il est difficile de comparer tous ces outils parce qu'ils implémentent des optimisations différentes, produisent différents types d'automates, et ne sont pas exécutés de la même façon (un script est plus lent qu'un exécutable). En ce qui concerne le temps, précisons que l'horloge de machine sur laquelle ont été exécutés les tests a une résolution de 0.01s.

[9] Nous attirons l'attention du lecteur sur le fait qu'LBTT et LBT sont deux programmes différents aux noms proches. Le premier est un testeur de traducteurs, le dernier est un traducteur.

Comme les temps affichés sont la somme des mesures des 100 (ou 94) exécutions de chaque outil, des valeurs inférieures à la seconde indiquent donc que des exécutions ont été mesurées avec un temps nul.

Nous pouvons cependant tirer quelques conclusions:

LTL2BA peut être comparé avec Spot/FM+(L) car ils utilisent alors les mêmes règles de simplification de formules. Dans le cas généralisé, Spot/FM+(L) est supérieur à LTL2BA. Cela tient à notre avis au fait que les successeurs sont calculés avec des implicants premiers. Dans le cas dégénéralisé, LTL2BA est meilleur: ses simplifications visant à retirer les conditions d'acceptation inutiles (en étudiant les composantes fortement connexes) lui permettent de dégénéraliser l'automate plus facilement.

Les réécritures de formule (15) confirment leur supériorité sur (123) et (L), sauf dans un cas: dans les automates généralisés, Spot/FM+(L) est meilleur que Spot/FM+(15). Ceci indique peut être qu'il manque une entrée dans nos règles pour (1) ou (5).

L'optimisation du déterminisme peut effectivement être constaté en comparant la ligne Spot/FM+(15) avec la ligne Spot/FM+(15)+(D).

Le nombre de transitions des produits est directement lié au temps que mettra un algorithme d'*emptiness check* à parcourir déterminer si l'automate est vide. De ce point de vue, les meilleurs construction sont systématiquement obtenues en activement cette optimisation.

Le *branching postponement* n'est pas forcément une bonne chose. Nous l'avions déjà montré sur un exemple particulier (figure 4.16 page 94), ces résultats le confirment à plus large échelle: les mesures pour Spot/FM+(15)+(D) sont souvent meilleures que celles pour Spot/FM+(15)+(D)+(B).

Il pourrait être intéressant de calculer systématiquement les deux automates (avec ou sans *branching postponement*) et de retenir celui qui semble le meilleur.

L'utilisation des relations de simulation permet de réduire les automates de façon assez importante. C'est flagrant en comparant les lignes avec ou sans (S).

Le traducteur de Spin n'est pas très efficace et nous gagnerions à ne plus l'utiliser. Plusieurs des outils listés lui sont supérieurs et peuvent produire une clause *never claim*, c'est-à-dire un automate de Büchi dans le format d'entrée de Spin.

4.5 Une idée: prendre en compte la logique des promesses

Une optimisation que nous n'avons pas eu le loisir de mettre en pratique serait d'utiliser la sémantique des promesses pour en limiter le nombre lors de la traduction.

Considérons la formule LTL $a \mathsf{U} \mathsf{F} b$. Les réécritures suivantes permettent de construire l'automate de la figure 4.17a page suivante, où « $\circ$ » représente le complément de $\mathsf{P} \mathsf{F} b$ et « $\bullet$ » celui de $\mathsf{P} b$.

$$B(a \mathsf{U} \mathsf{F} b) = \mathrm{Var}[b] \vee (\mathrm{Prom}[b] \wedge \mathrm{Next}[\mathsf{F} b]) \vee (\mathrm{Var}[a] \wedge \mathrm{Prom}[\mathsf{F} b] \wedge \mathrm{Next}[a \mathsf{U} \mathsf{F} b]) \quad (4.9)$$

$$B(\mathsf{F} b) = \mathrm{Var}[b] \vee (\mathrm{Prom}[b] \wedge \mathrm{Next}[\mathsf{F} b]) \quad (4.10)$$

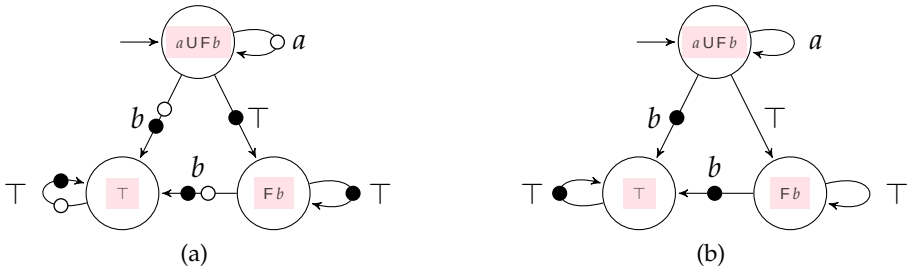


FIG. 4.17: Automates pour $a \mathsf{U} \mathsf{F} b$.

Jusqu'à présent, les promesses nous ont uniquement servi de marqueurs pour indiquer quelles étaient les portions de l'automate dans lesquelles il ne devait pas rester de façon continue. Ces promesses sont générées automatiquement à partir des règles de tableau de la figure 4.1 page 75 quand le fils d'une formule remet à plus tard la vérification d'une formule qui *doit* finir par être vérifiée (typiquement, pour reconnaître $a \mathsf{U} b$ il ne faut pas vérifier uniquement a en remettant sans cesse à plus tard la vérification de b: la traduction promet de vérifier b). Une autre façon d'expliquer l'effet de ces promesses consiste à dire que si $\mathsf{P}\,\varphi$ est porté par un arc, cela signifie que la formule $\mathsf{F}\,\varphi$ doit être vérifiée à partir de cet arc. Nous en déduisons que deux promesses $\mathsf{P}\,\varphi$ et $\mathsf{P}\,\psi$ sont équivalentes si $\mathscr{L}(\mathsf{F}\,\varphi) = \mathscr{L}(\mathsf{F}\,\psi)$.

En particulier comme $\mathscr{L}(\mathsf{F}\,\mathsf{F}\,b) = \mathscr{L}(\mathsf{F}\,b)$ nous avons $\mathsf{P}\,\mathsf{F}\,b \equiv \mathsf{P}\,b$ et l'équation (4.9) peut alors se réduire à

$$B(a \mathsf{U} \mathsf{F} b) = \mathrm{Var}[b] \vee (\mathrm{Prom}[b] \wedge \mathrm{Next}[\mathsf{F}\,b]) \vee (\mathrm{Var}[a] \wedge \mathrm{Prom}[b] \wedge \mathrm{Next}[a \mathsf{U} \mathsf{F} b])$$

L'automate construit à partir de cette écriture est celui de la figure 4.17b. Parce qu'elle est faite lors de la traduction d'une expression booléenne en BDD, cette réécriture peut potentiellement permettre des simplifications de cette expression. Il serait aussi possible de tirer parti des implications entre promesses: si la promesse $\mathsf{P}\,\mathsf{G}\,a$ est tenue, alors $\mathsf{P}\,a$ l'est nécessairement. À notre avis il y a là une *logique des promesses* qui mériterait d'être étudiée.

4.6 Conclusion

Nous avons présenté les divers algorithmes existants de traduction de formules LTL en automate de Büchi. La contribution de ce chapitre dont nous sommes le plus fier est l'illustration de la méthode tableau qui unifie la traduction d'une formule LTL en automates de Büchi basés sur les états ou sur les transitions (figures 4.8 page 79 et 4.9 page 81). Cette représentation graphique du tableau, combinée à l'utilisation des promesses, rend la construction extrêmement simple à expliquer et comprendre.

Nous avons proposé de nouvelles règles de simplifications de formules LTL basées sur l'inclusion de langages, utilisant les algorithmes de traduction de formule LTL et de test

de vacuité présentés dans ce document. Ces règles, combinées à quelques réécritures, offrent une meilleure réduction des formules et des automates les représentant que les autres règles publiées jusqu'à présent, mais au prix d'une complexité plus forte. Nous avons argué que cette complexité n'était pas un problème en pratique car le temps (négligeable) passé à réduire la formule sera très largement compensé par le temps (non négligeable) gagné à explorer un automate produit plus petit.

Nous avons combiné l'algorithme de traduction de Couvreur [34] avec diverses optimisations inspirées de la littérature. Ces optimisations rendent la traduction compétitive même vis-à-vis d'outils comme LTL2NBA, qui utilisent des post-traîtements simplificateurs supérieurs.

Enfin, nos expérimentations donnent la mesure du gain apporté par l'emploi des TGBA lorsque la présence des conditions d'acceptation tient uniquement à la formule LTL: celui-ci reste modéré car la moyenne du nombre de conditions d'acceptation par automate est inférieure à 2. Dans le chapitre 7 nous montrerons que la prise en compte d'hypothèses d'équité fera exploser ce facteur en ajoutant des conditions d'acceptation dans l'automate représentant le modèle.

Chapitre 5

Emptiness checks de TGBA

Ce chapitre est basé sur des travaux présentés avec Jean-Michel Couvreur et Denis Poitrenaud au workshop SPIN 2005 [37]. Il brosse un panorama des différents algorithmes d'*emptiness check* pour automates de Büchi (généralisés ou non) et propose différentes optimisations et heuristiques.

5.1 Introduction

L'*emptiness check* est l'un des algorithmes clefs de l'approche du *model checking* basée sur les automates introduite section 2.8 page 28. Son rôle est de déterminer si l'automate résultant du produit du modèle et de la négation de la propriété à y vérifier accepte une exécution. Autrement dit, on souhaite savoir si le langage de l'automate est vide ou non. S'il est vide, cela signifie qu'il n'existe pas d'exécution du modèle qui invalide la propriété: cette dernière est donc vérifiée. S'il est non-vide, il existe alors une exécution du système qui invalide la propriété ; cette exécution est appelée *contre-exemple*.

L'automate « produit » dont on veut tester la vacuité est généralement beaucoup plus gros que le modèle et la formule à partir desquels il est construit. On préférera donc éviter de le construire en entier. Les algorithmes d'*emptiness check* dits « à la volée » sont ceux qui explorent l'automate de façon à ce que celui-ci puisse être construit au fur et à mesure de son exploration. Ce n'est pas l'*emptiness check* qui est « à la volée » à proprement parler, mais la construction de l'automate exploré par l'algorithme. Outre la possibilité de travailler sur les automates qui ne tiendraient pas forcément en mémoire, nous gagnons aussi du temps si l'algorithme est capable de répondre avant d'avoir exploré tout l'automate ; par exemple dès que l'algorithme a déterminé l'existence d'un contre-exemple.

5.2 Principe et historique

Étant donné un TGBA $A = \langle AP, Q, Q^0, \mathcal{F}, \delta \rangle$, un algorithme d'*emptiness check* doit déterminer si $\mathscr{L}(A) = \emptyset$ selon la définition 28 page 49.

	non généralisé	généralisé
NDFS	Ⓢ Courcoubetis et al. [32, 33]CVWY90 Ⓢ Godefroid et Holzmann [65] Ⓢ Holzmann et al. [78] Ⓢ Gastin et al. [57] Ⓢ Schwoon et Esparza [106, 107]SE05	Ⓖ Tauriainen [119, 121] Ⓣ Tauriainen [120] Tau03 Ⓣ Couvreur et al. [37] Tau03 opt Ⓣ Tauriainen [122, 123]
CFC	Ⓢ Geldenhuys et Valmari [59] GV04	Ⓖ Lichtenstein et Pnueli [92] Ⓣ Couvreur [34] Cou99 Ⓖ Geldenhuys et Valmari [60] Ⓖ Hammer et al. [70] Ⓣ Couvreur et al. [37] Cou99 shy Ⓣ Li et al. [91]

TAB. 5.1: Classification des algorithmes d'*emptiness check* selon leur principe (NDFS ou calcul des CFC) et le type de conditions d'acceptation de Büchi supportées (généralisées ou non). La signification de Ⓢ, Ⓖ et Ⓣ est indiquée dans la figure 5.1 page suivante. Les abréviations en sans-sérif (CVWY90, Cou99, ...) nous serviront à faire référence à ces algorithmes dans les tableaux comparatifs.

Supposons que la réponse soit négative, c'est-à-dire qu'il existe un chemin acceptant $\sigma \in \mathrm{Acc}(A)$. D'après la proposition 1 page 50, il existe un chemin de la forme

$$\sigma = \sigma(0) \cdot \sigma(1) \cdots \sigma(i-1) \cdot (\sigma(i) \cdot \sigma(i+1) \cdots \sigma(i+k-1))^{\omega}$$

sachant que $\bigcup_{j=i}^{i+k-1} \sigma(j)^{\mathrm{acc}} = \mathcal{F}$. Cela signifie qu'il existe un état accessible $\sigma(i)^{\mathrm{in}}$ de l'automate autour duquel on peut trouver un circuit traversant toutes les conditions d'acceptation.

Tous les algorithmes d'*emptiness check* recherchent un tel circuit et l'on peut distinguer deux méthodes:

– Il y a tout d'abord ceux qui s'appuient sur des parcours en profondeur imbriqués: un premier parcours construit le préfixe, d'autres parcours en profondeur sont lancés à partir des états atteints pour trouver d'éventuels circuits acceptants. Nous ferons référence à cette classe d'algorithmes à travers l'acronyme NDFS, pour *nested depth-first search*. La section 5.3 page 108 leur est consacrée.
– Ensuite il y a ceux qui reposent sur un calcul des composantes fortement connexes (CFC) de l'automate à explorer. En effet, si une CFC contient des transitions étiquetées par toutes les conditions d'acceptation, alors elle contient un circuit acceptant. Cette classe d'algorithmes, sur laquelle s'appuient les chapitres suivants, sera abordée section 5.4 page 118.

La figure 5.1 page ci-contre montre une généalogie de ces différents algorithmes. Il faut noter que tous n'ont pas été présentés pour les mêmes types d'automates. La différence entre étiquetage sur les états ou transitions n'est pas très importante: il est relativement aisé de transformer un *emptiness check* sur les états en une version sur les transitions (et l'opération inverse est triviale). En revanche la prise en compte des conditions d'acceptation généralisées (Ⓖ et Ⓣ) ou non (Ⓢ) constitue une différence importante entre ces algorithmes. Nous pouvons donc catégoriser ces algorithmes sous la forme du tableau 5.1.

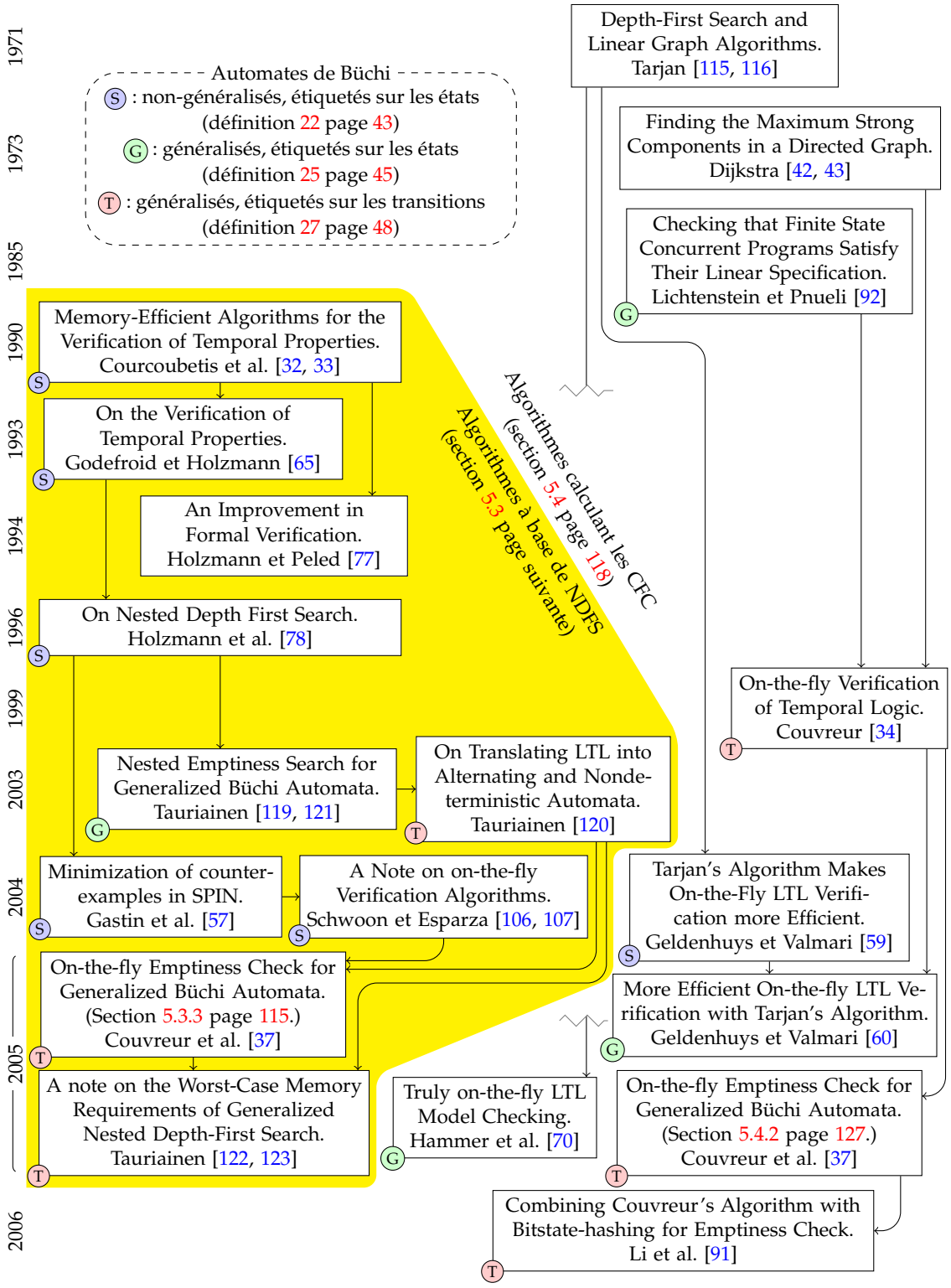


FIG. 5.1: Généalogie des algorithmes d'*emptiness check*.

Schwoon et Esparza [106, 107] ont comparé les algorithmes NDFS à ceux basés sur les CFC en les appliquant à des automates de Büchi avec condition d'acceptation unique sur les états (définition 22 page 43). Leurs conclusions sont les suivantes:
– l'algorithme de Couvreur [34] est le meilleur de ceux qui calculent les CFC,
– l'algorithme de Schwoon et Esparza [107] est le meilleur de ceux basés sur les NDFS,
– lorsque l'automate de Büchi est faible[1], un unique parcours en profondeur est nécessaire [25] ; sinon les algorithmes basés sur les CFC devraient être préférés aux NDFS, exception faite des cas où l'on utilise une technique de *bit-state hashing* (que nous aborderons section 5.7 page 140).

Dans ce chapitre, nous évaluerons ces algorithmes sur des TGBA (définition 27 page 48) pour mettre en avant les avantages des algorithmes d'*emptiness check* supportant des conditions d'acceptation généralisées (colonne du droite du tableau 5.1 page 106) sur ceux, plus traditionnels, qui ne supportent qu'une condition d'acceptation (colonne de gauche). D'autre part nous introduirons plusieurs variantes, certaines étant des optimisations, d'autres des heuristiques.

5.3 Parcours en profondeur imbriqués

Les algorithmes NDFSs furent initialement développés pour les automates de Büchi avec une seule condition d'acceptation sur les états [32]. De façon très simple, un algorithme NDFS va effectuer une premier parcours en profondeur à partir de l'état initial q^0 de l'automate jusqu'à ce qu'il trouve un état acceptant s dont il a déjà visité tous les descendants. À partir de cet état s l'algorithme débute un second parcours en profondeur (imbriqué dans le premier) pour déterminer si l'état s est accessible depuis lui-même. Le second NDFS est lancé dans l'ordre postfixe des découvertes d'états acceptants, afin de s'assurer que chaque état ne sera visité que deux fois dans le pire des cas.

Cet algorithme fut ensuite amélioré de façon à ce que les deux parcours en profondeur puissent partager la même table de hachage recensant les états visités [65], ainsi que pour quitter plus tôt tout en supportant des réduction basées sur les ordres partiels [78].

L'algorithme d'Holzmann et al. [78] fut successivement raffiné par Gastin et al. [57] et Schwoon et Esparza [107]. Les premiers montrent qu'avec un bit supplémentaire par état il est possible de terminer l'algorithme plus tôt ; les derniers montrent qu'on peut en faire autant sans ce bit supplémentaire.

En parallèle, Tauriainen a généralisé cet algorithme pour supporter plusieurs conditions d'acceptation sur les états [121], ou sur les transitions [120]. Passer des états aux transitions est aisé, le véritable défi était de trouver un moyen de gérer plusieurs conditions d'acceptation. L'idée de Tauriainen est de répéter le parcours en profondeur interne plusieurs fois (au pire $|\mathcal{F}|$ fois) et de conserver sur chaque état la liste des conditions d'acceptation qu'il est possible de traverser entre la pile de recherche et cet état.

[1] Un automate de Büchi (non généralisé) est faible si toutes ses composantes fortement connexes maximales contiennent uniquement des états acceptants ou uniquement des états non acceptants. Černá et Pelánek [25] montrent les formules LTL persistantes, dont la classe est définie syntaxiquement par Chang et al. [26], peut être traduite en un automates de Büchi faibles.

5.3.1 Le cas non généralisé

La figure 5.2 page suivante présente une version récursive de l'algorithme de Schwoon et Esparza [107], adaptée aux automates étiquetés sur les transitions. Nous appellerons cet algorithme SE05. Dans un premier temps, on ignorera le second paramètre de `dfs_blue` ainsi que la boîte de la ligne 15: il s'agit d'une optimisation qui fut initialement proposée par Schwoon et Esparza et que nous généraliserons section 5.3.2.

Comme expliqué précédemment, cet algorithme fonctionne sur des automates n'utilisant qu'une condition d'acceptation. Le premier parcours en profondeur, `dfs_blue`, cherche des transitions acceptantes et lance à partir de celles-ci (dans l'ordre postfixe de leurs découvertes, c'est-à-dire après avoir visité tous les successeurs de ces transitions) un parcours en profondeur imbriqué `dfs_red`. Les états sont coloriés de quatre façons à l'aide du tableau associatif H:

- Les états blancs n'ont pas encore été visités.
  Dans l'algorithme cela se traduit par $s \notin H$.
- Les états cyans sont ceux qui se trouvent sur la pile de recherche de `dfs_blue`.
  $H[s] = \mathsf{cyan}$.
- Les états bleus sont ceux qui ont été visités par `dfs_blue`, ne se trouvent plus sur sa pile de recherche, mais n'ont pas été visités par `dfs_red`
  $H[s] = \mathsf{blue}$.
- Les états rouges sont ceux qui ont été visités par `dfs_red`
  $H[s] = \mathsf{red}$.

Comme tous les états entre l'état initial et celui sur lequel est lancé `dfs_red` sont dans la pile de recherche de `dfs_blue`, ils sont cyans. Si `dfs_red` parvient à atteindre un état cyan, cela signifie que la transition acceptante à partir de laquelle il a été lancé est accessible. Il existe donc un circuit acceptant et l'algorithme peut retourner $\bot$ immédiatement.

Si `dfs_blue` franchit une transition acceptante qui l'amène sur un état cyan, il peut lui aussi signaler l'existence d'un circuit acceptant immédiatement. Ce test est fait ligne 15, mais techniquement il est superflu (rappelons que nous ignorons toujours la boîte pour le moment): si l'on omet les lignes 15–16, le test sera fait ligne 23 dans `dfs_red`.

La figure 5.3 page 111 illustre le déroulement de cet algorithme sur un petit automate. Les états y sont coloriés avec les couleurs listées ci-dessus. Les flèches en pointillés indiquent les transitions non explorées ; les flèches épaisses symbolisent le chemin de recherche des parcours en profondeur. Dans les états s_2 et s_5 les transitions sortantes seront parcourues de droite à gauche.

Dans un premier temps, `dfs_blue` descend dans l'automate en traversant les états s_1, s_2, s_3, s_4 jusqu'à retomber sur l'état s_3 déjà vu (a). L'algorithme dépile donc des états, en les étiquetant bleus, jusqu'à soit avoir de nouveaux successeurs à explorer, soit revenir d'une transition acceptante (b). Comme l'algorithme vient de traiter tous les successeurs de la transition acceptante $s_2 \to s_3$, un `dfs_red` est lancé à la recherche d'un état cyan. Cette recherche échoue (c), mais tous les états ont étés marqués comme rouges au passage. Le parcours `dfs_blue` continue ensuite à partir de l'état s_2 où il s'était arrêté, jusqu'à atteindre l'état s_4 (d). s_4 est rouge, donc on sait qu'aucun circuit acceptant n'y passe et il peut-être ignoré. `dfs_blue` continue son parcours vers s_1 (e), qu'il a déjà vu, avant

```
1  Entrées : Un TGBA A = ⟨AP, Q, Q^0, F, δ⟩ tel que |F| = 1
2  Résultat : ⊤ si et seulement si L(A) = ∅
3  Données :  H: map of Q ↦ {cyan, blue, red}
4  begin
5      foreach q^0 ∈ Q^0 do
6          if dfs_blue(q^0, ∅) = ⊥ then
7              return ⊥
8      return ⊤
9  end
10 dfs_blue(s ∈ Q, Acc ⊆ F)
11     H[s] ← cyan
12     foreach ⟨q, p, a, d⟩ ∈ δ such that q = s do
13         if d ∉ H then
14             if dfs_blue(d, a) = ⊥ then return ⊥
15         else if H[d] = cyan ∧ (a = F ∨(Acc = F ∧ s ≠ d)) then
16             return ⊥
17         if a = F then
18             if dfs_red(d) = ⊥ then return ⊥
19     H[s] ← blue
20     return ⊤
21 end
22 dfs_red(s ∈ Q)
23     if H[s] = cyan then
24         return ⊥
25     H[s] ← red
26     foreach ⟨q, p, a, d⟩ ∈ δ such that q = s do
27         if H[d] = blue then
28             if dfs_red(d) = ⊥ then return ⊥
29     return ⊤
30 end
```

FIG. 5.2: L'algorithme SE05 de Schwoon et Esparza [107].

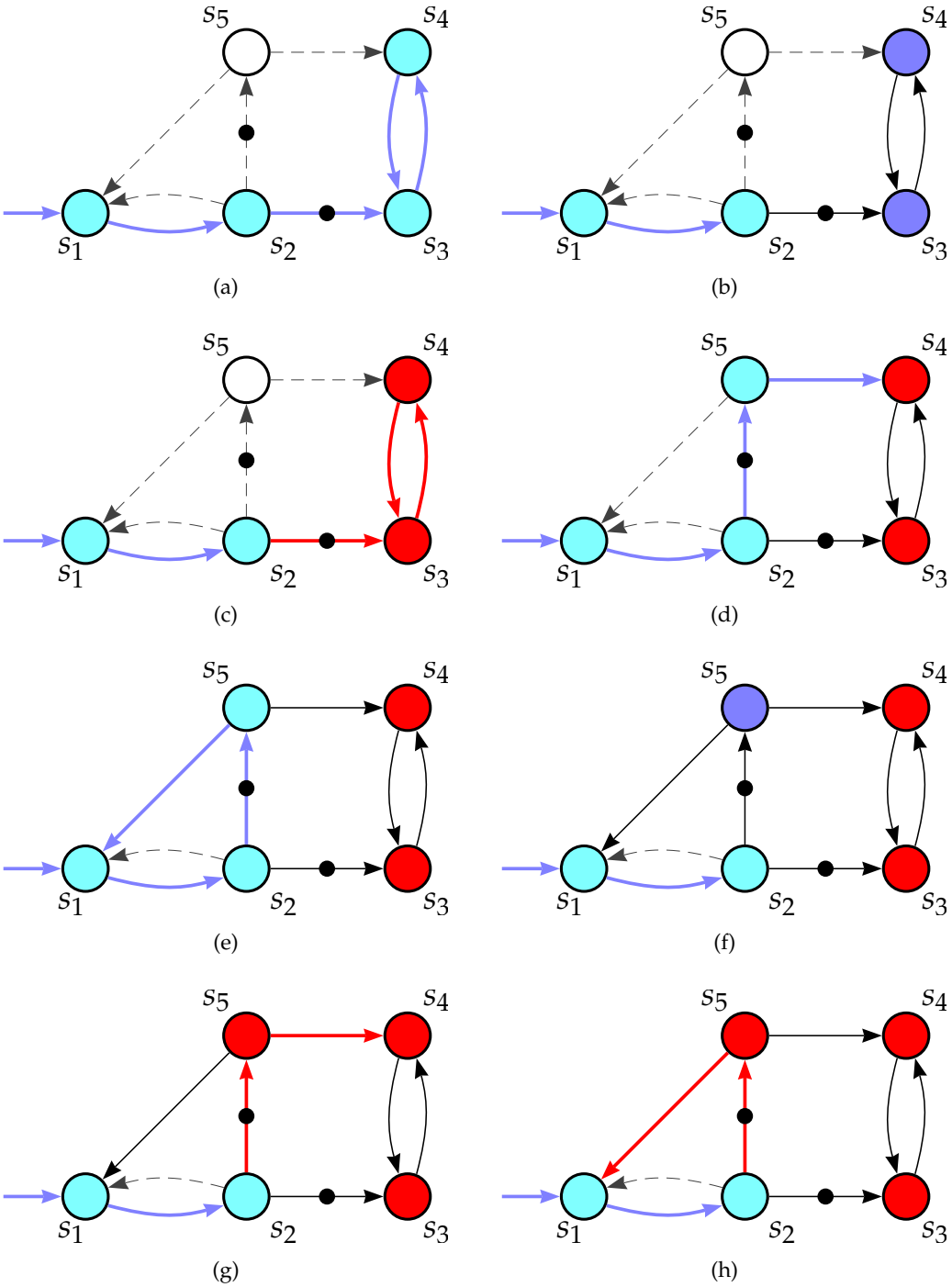


FIG. 5.3: Déroulement de l'algorithme SE05. L'algorithme s'arrête à l'étape (e) avec la boîte de la ligne 15 de la figure 5.2 page ci-contre ; sans elle il se poursuit jusque (h).

de revenir sur ses pas à nouveau jusqu'en s_2 (f). Comme tous les successeurs de la transition acceptante ont été traités, un `dfs_red` est à nouveau lancé: il ignore s_4 lorsqu'il atteint (g), et peut retourner $\bot$ (l'automate n'est pas vide) lorsqu'il atteint enfin un état cyan (h). La transition $s_2 \to s_1$ n'aura pas été explorée.

La découverte d'un état cyan signifie toujours qu'il existe un circuit accessible, mais pour que cet algorithme puisse conclure que le langage n'est pas vide, il faut que le circuit contienne une transition acceptante. Si l'état cyan est trouvé par `dfs_red` on sait que c'est le cas, car `dfs_red` n'est lancé que depuis une transition acceptante. De même, `dfs_blue` peut aussi facilement conclure s'il trouve un état cyan en franchissant une transition acceptante. Schwoon et Esparza ont noté que `dfs_blue` pouvait aussi conclure à un circuit si les deux conditions suivantes étaient réunies:
– à partir d'un état s, `dfs_blue` découvre un successeur cyan qui n'est pas s
– la transition franchie pour arriver à s est acceptante
Ces conditions sont vérifiées dans la figure 5.3e et sont testées par la boîte de la ligne 15 page 110.

Naturellement, si l'on déplace la condition d'acceptation de la transition $s_2 \to s_5$ pour la mettre sur la transition $s_1 \to s_2$, cette optimisation ne fonctionne plus et le circuit ne sera détecté que par `dfs_red`.

On pourrait alors ajouter un nouveau test: si l'avant dernière transition est acceptante et que la prochaine transition amène sur un état cyan qui n'est ni l'état courant ni le dernier, alors il existe un circuit acceptant.

On peut continuer ainsi sur plusieurs profondeurs, mais cela n'est pas satisfaisant. La section qui suit généralise cette technique.

5.3.2 Ajout des poids

Lors du workshop SPIN'05 [37], nous avons proposé d'utiliser des poids pour généraliser l'optimisation de Schwoon et Esparza dans `dfs_blue`. En réalité, nous avons implémenté cette optimisation de Tau03, l'algorithme de Tauriainen [120] travaillant sur des automates généralisés, mais il est plus simple de commencer par la présenter dans le cas non-généralisé, c'est-à-dire sur SE05.

L'idée est de numéroter tous les états cyans (et seulement ceux-ci) par le nombre de transitions acceptantes franchies depuis l'état initial (ligne 12. Si `dfs_blue` atteint un état cyan dont le poids est plus faible que celui dont il vient (ligne 17), c'est qu'il existe une transition acceptante entre les deux, et donc un circuit acceptant dans l'automate.

La figure 5.4 page ci-contre montre l'algorithme ainsi modifié, et la figure 5.5 page 114 illustre son déroulement sur notre exemple. Il faut noter qu'à la différence de l'original ce nouvel algorithme s'arrêterait aussi rapidement si la condition d'acceptation de la transition $s_2 \to s_5$ avait été déplacée sur la transition $s_1 \to s_2$.

```
1  Entrées : Un TGBA A = ⟨Σ, Q, Q^0, F, δ⟩ tel que |F| = 1
2  Résultat : ⊤ si et seulement si L(A) = ∅
3  Données :  H: map of Q ↦ {cyan, blue, red}
              W: map of Q ↦ ℕ
4  begin
5      foreach q^0 ∈ Q^0 do
6          if dfs_blue(q^0, 0) = ⊥ then
7              return ⊥
8      return ⊤
9  end
10 dfs_blue(s ∈ Q, w ∈ ℕ)
11     H[s] ← cyan
12     W[s] ← w
13     foreach ⟨q, p, a, d⟩ ∈ δ such that q = s do
14         n ← (a = F) ? w + 1 : w
15         if q ∉ H then
16             if dfs_blue(d, n) = ⊥ then return ⊥
17         else if H[d] = cyan ∧ n > H[d] then
18             return ⊥
19         if a = F then
20             if dfs_red(d) = ⊥ then return ⊥
21     delete W[s]
22     H[s] ← blue
23     return ⊤
24 end
25 dfs_red(s ∈ Q)
26     if H[s] = cyan then
27         return ⊥
28     H[s] ← red
29     foreach ⟨q, p, a, d⟩ ∈ δ such that q = s do
30         if H[d] = blue then
31             if dfs_red(d) = ⊥ then return ⊥
32     return ⊤
33 end
```

FIG. 5.4: L'algorithme SE05 modifié pour utiliser les poids. Les modifications apparaissent en couleur.

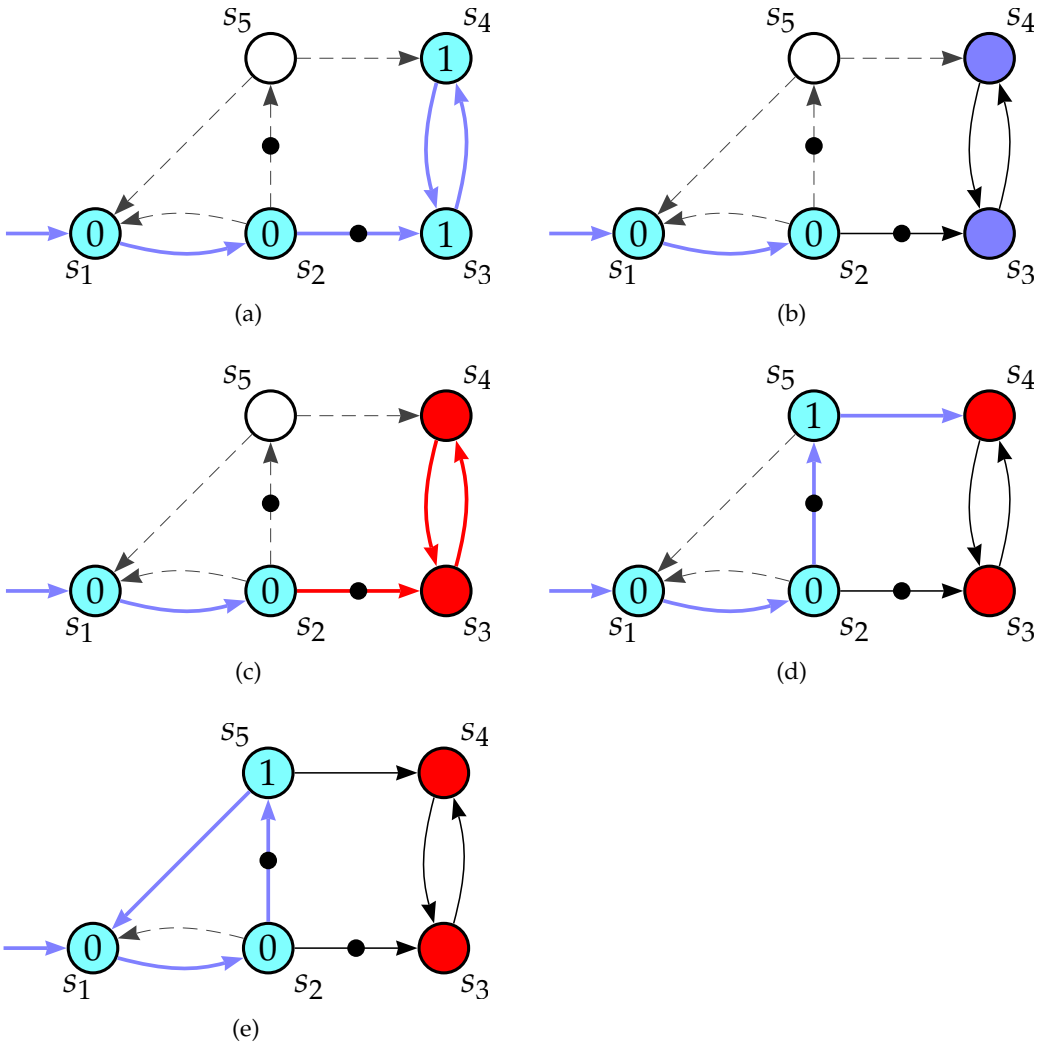


FIG. 5.5: Utilisation des poids lors du premier parcours en profondeur.

5.3.3 Cas généralisé

Dans le cas général où le TGBA possède plusieurs conditions d'acceptation, les algorithmes précédents ne peuvent pas être appliqués directement. L'automate doit d'abord être dégénéralisé (section 3.3.6 page 55) ce qui peut multiplier sa taille par le nombre de conditions d'acceptation.

Il existe un algorithme d'*emptiness check* NDFS fonctionnant directement sur des automates généralisés. Cet algorithme, développé par Tauriainen été initialement destiné aux automates de Büchi généralisés étiquetés sur les états [119, 121] puis a été adapté aux transitions [120]. Malheureusement, cette adaptation, un peu trop proche de la version sur les états, manque une optimisation importante: le parcours en profondeur descend récursivement dans *tous* les successeurs même s'il peut répondre après avoir visité le premier. La version que nous proposons figure 5.6 page suivante corrige ce léger problème ; les lignes colorées correspondent à la mise en œuvre des poids et peuvent être ignorées pour le moment.

Comme tous les NDFS, cet algorithme utilise deux parcours en profondeur imbriqués.

Les états cyans correspondent toujours aux états de la pile de recherche du premier parcours, et les états bleus sont ceux qui ont été visités par le premier parcours sans être sur la pile de recherche. Cette distinction est une autre optimisation apportée à l'agorithme (l'original n'utilise pas d'états cyans). En plus de ces deux couleurs, l'algorithme étiquette les états par des conditions d'acceptation. Lorsque le premier parcours en profondeur (`dfs_blue`) revient d'une transition portant un ensemble de conditions d'acceptation a, il lance un second parcours (`dfs_red`, ligne 23) qui va étiqueter avec ces conditions d'acceptation tous les états accessibles depuis cette transition. Ce second parcours ne descend récursivement (ligne 33) que dans les états qui ne sont pas déjà étiquetés par les conditions d'acceptation qui l'ont démarré.[2] Avec cette condition d'arrêt, un état peut être visité par `dfs_red` au plus autant de fois qu'il existe de conditions d'acceptation. Si ce second parcours parvient à amener un ensemble de conditions d'acceptation a sur un état cyan (ligne 31), c'est qu'il existe un circuit autour de ces états visitant ces conditions d'acceptation. L'algorithme peut conclure à l'existence d'un chemin acceptant dès qu'il étiquette un état cyan par $\mathcal{F}$. (Dans l'algorithme original qui n'utilise pas d'état cyan, la conclusion se fait un peu plus tard lorsque l'algorithme parvient à étiqueter par $\mathcal{F}$ l'état à partir duquel `dfs_red` est été initié.)

La figure 5.7 page 117 illustre le fonctionnement de cet algorithme sur un exemple légèrement différent des précédents: l'automate possède ici deux conditions d'acceptation.

L'algorithme commence par exécuter `dfs_blue` récursivement sur $s_1 \to s_2 \to s_3 \to s_4 \to$ jusqu'à atteindre l'état s_3 qu'il a déjà vu (a). Il revient alors en arrière jusqu'à ce qu'il ait de nouveaux successeurs à visiter, ou qu'il revienne sur une transition étiquetée par des conditions d'acceptation (b). Comme la transition $s_2 \to s_3$ est étiquetée par « ●○ », un parcours `dfs_red` est lancé pour propager cette étiquette sur tous les états accessibles qui ne l'ont pas (c). Ce parcours en profondeur se termine sans avoir étiqueté d'état cyan, donc `dfs_blue` reprend où il en était en visitant s_5, s_4 (d) et s_1 (e) tous les deux déjà

[2] Dans l'algorithme SE05, c'étaient les états rouges qui stoppaient les parcours en profondeur. Ici, nous pouvons considérer qu'il y a autant de teintes de rouge différentes qu'il y a de conditions d'acceptation.

```
1  Entrées : Un TGBA A = ⟨Σ, Q, Q^0, F, δ⟩ tel que |F| > 1
2  Résultat : ⊤ si et seulement si L(A) = ∅
3  Données :  H: map of Q ↦ ⟨color ∈ {cyan, blue}, acc ⊆ F⟩
              W: map of Q ↦ (map of F ↦ ℕ)
              weight: map of F ↦ ℕ
4  begin
5      foreach f ∈ F do
6          weight[f] ← 0
7      foreach q^0 ∈ Q^0 do
8          if dfs_blue(q^0)= ⊥ then
9              return ⊥
10     return ⊤
11 end
12 dfs_blue(s ∈ Q)
13     H[s] ← ⟨cyan, ∅⟩
14     W[s] ← weight
15     foreach ⟨l, a, t⟩ such that ⟨s, l, a, t⟩ ∈ δ do
16         if t ∉ H then
17             foreach f ∈ a do
18                 weight[f] ← weight[f] + 1
19             if dfs_blue(t) = ⊥ then
20                 return ⊥
21             foreach f ∈ a do
22                 weight[f] ← weight[f] − 1
23         if dfs_red(s, H[s].acc ∪ a, t) = ⊥ then
24             return ⊥
25     delete W[s]
26     H[s].color ← blue
27     return ⊤
28 end
29 dfs_red(s ∈ Q, Acc ⊆ F, t ∈ Q)
30     ⟨tcol, tacc⟩ ← H[t]
31     if tcol = cyan ∧ F = (H[s].acc ∪ Acc ∪ tacc ∪ {f ∈ F | weight[f] > W[t][f]}) then
32         return ⊥
33     else if Acc ⊈ tacc then
34         H[t].acc ← tacc ∪ Acc
35         foreach ⟨l, a, t⟩ such that ⟨s, l, a, t⟩ ∈ δ do
36             if t ∈ H ∧ dfs_red(s, Acc, t) = ⊥ then
37                 return ⊥
38     return ⊤
39 end
```

FIG. 5.6: Tau03 opt: Variation sur l'algorithme d'*emptiness check* de Tauriainen [120]. Les lignes colorées sont facultatives et sont une implémentation des poids.

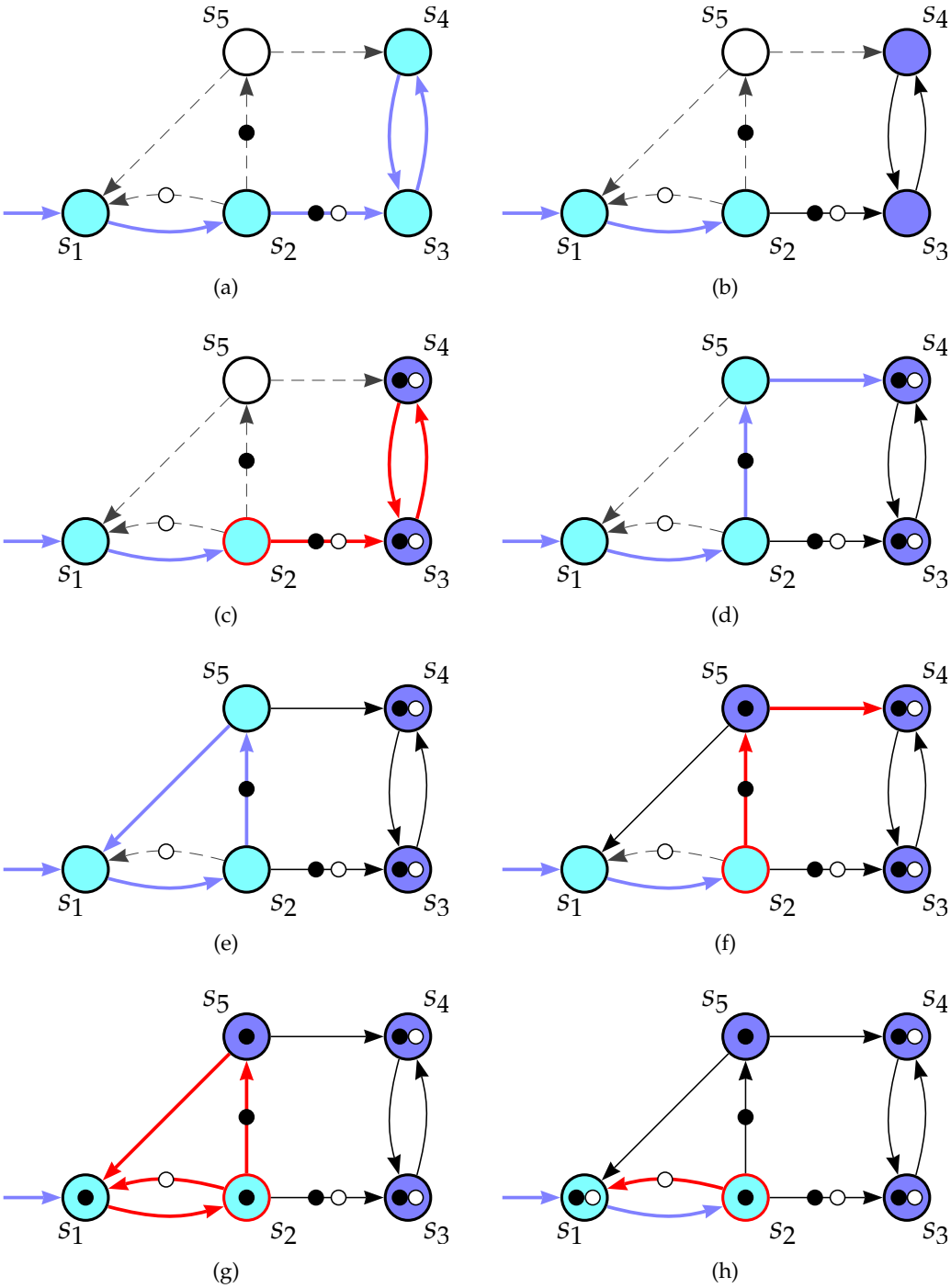


FIG. 5.7: Déroulement de l'algorithme Tau03.

vus. L'algorithme rebrousse maintenant chemin jusqu'à l'état s_2 ; comme la transition $s_2 \rightarrow s_5$ est étiquetée par une condition d'acceptation, un `dfs_red` est lancé pour la propager. Celui-ci commence par étiqueter s_5, s'arrête sur l'état s_4 qui la possède déjà (f), puis il étiquette les états s_1, s_2, puis termine sa course en repassant sur s_1 qu'il a déjà étiqueté (g). À nouveau, cet appel à `dfs_red` n'est pas parvenu à étiqueter un état cyan par $\mathcal{F}$, le `dfs_blue` reprend donc sa course où il en était: en s_2. Il reste la transition $s_2 \rightarrow s_1$ à explorer, elle amène sur un état que le parcours bleu à déjà visité, mais lorsque le `dfs_red` est lancé pour propager l'étiquette de cette transition, cela étiquette s_1 par $\mathcal{F}$ (h) et permet à l'algorithme de conclure qu'il existe un chemin acceptant dans l'automate.

Si l'automate à tester possède n états et $m = |\mathcal{F}|$ conditions d'acceptation, cet algorithme peut donc visiter $n(m+1)$ états dans le pire cas (chaque état est vu au plus 1 fois par `dfs_blue` et m fois par `dfs_red`). L'application sur le même automate, d'un algorithme tel que SE05 qui demande une dégénéralisation demanderait la visite de $2nm$ états dans le pire cas.

Bien que ce ne soit pas visible sur cet exemple, l'algorithme peut être amélioré en utilisant des poids de la même façon que nous l'avons fait pour SE05 section 5.3.2 page 112. Cependant il faudra ici utiliser un compteur différent pour chaque condition d'acceptation. Les lignes colorées de la figure 5.6 page 116 apportent les modifications nécessaires. Lors du Workshop SPIN'05, Willem Wisser a suggéré une implémentation légèrement différente où chaque état serait numéroté par sa profondeur dans la pile de `dfs_blue`, et sur $|\mathcal{F}|$ piles séparées, ont garderait trace de la profondeur de la dernière transition étiquetée par la transition correspondante. Une telle implémentation occuperait moins de mémoire.

Pour les preuves des deux algorithmes NDFS que nous avons présentés dans les figures 5.4 page 113 et 5.6 page 116 nous renvoyons le lecteur aux articles de Schwoon et Esparza [106, 107] et Tauriainen [122, 123]. Ce dernier intègre la correction que nous avons signalée au début de cette section pour signaler la détection d'un circuit acceptant plus tôt. Ces preuves n'incluent pas la gestion des poids, mais il est aisé de voir que:
- l'ajout des poids ne modifie ni l'ordre de parcours, ni la mise à jour des autres structures de l'algorithme
- les poids comptent le nombre d'occurences de chaque condition d'acceptation sur la pile de recherche
- lorsque le parcours en profondeur atteint un état de la pile de recherche, il peut à partir des poids calculer l'ensemble des conditions d'acceptation traversées depuis cet état et éventuellement conclure à l'existence d'un chemin acceptant.

Ces poids pourraient être utilisés avec n'importe quel *emptiness check* basés sur un parcours en profondeur.

5.4 Utilisation des composantes fortement connexes

Dans un automate, on appelle *composante fortement connexe* (CFC) un ensemble d'états dont tout état est accessible à partir de n'importe quel autre état distinct de cet ensemble. Un ensemble d'états formant un circuit est une composante fortement connexe. L'union de deux composantes fortement connexes partageant au moins un état est une compo-

sante fortement connexe. Un état seul, sans boucle sur lui-même, est un cas particulier de CFC dite *composante fortement connexe triviale*.

Une composante fortement connexe est *maximale* si elle n'est incluse (strictement) dans aucune autre CFC. Comme l'union de deux CFC partageant au moins un état est une CFC, les CFC maximales d'un automate forment une partition de ses états.

D'après la propriété 1 page 50, il existe un chemin acceptant dans un TGBA si et seulement s'il y existe un circuit acceptant et accessible.

Une autre stratégie pour déterminer l'absence d'un chemin acceptant est donc de calculer l'ensemble des CFC maximales. Si l'union de toutes les conditions d'acceptation d'une CFC non-triviale (mais pas nécessairement maximale) est l'ensemble $\mathcal{F}$ des conditions d'acceptation de l'automate, et que cette CFC est accessible depuis l'état initial q^0, alors on peut affirmer l'existence d'un chemin acceptant. À l'inverse si nous avons construit toutes les CFC maximales de l'automate mais qu'aucune d'elles ne couvre $\mathcal{F}$, alors on peut en déduire que l'automate ne contient aucun circuit acceptant, et donc aucun chemin acceptant.

Insistons sur le fait que nous devons construire l'*ensemble* des CFC maximales pour prouver que le langage d'un automate est vide, mais qu'il nous suffit de trouver une CFC non-triviale (pas nécessairement maximale) acceptante pour conclure qu'il ne l'est pas. Cela permet aux algorithmes d'*emptiness check* de répondre avant d'avoir exploré l'automate entier lorsque le langage de celui-ci n'est pas vide, et cela est un avantage lorsque l'automate exploré est construit à la volée.

L'idée de calculer les CFC maximales pour résoudre le problème de l'*emptiness check* remonte à 1985. L'algorithme de Lichtenstein et Pnueli [92] s'appuie sur le fait que tout automate contient au moins une CFC maximale sans arc sortant (dite *terminale*). Pour lister toutes les CFC maximales, il nous suffit donc de trouver une CFC maximale terminale, de la retirer de l'automate, puis de lister les CFC maximales de l'automate ainsi amputé. Pris au pied de la lettre, l'algorithme de Lichtenstein et Pnueli [92] est inefficace: il doit parcourir tout l'automate pour le partitionner en CFC maximales jusqu'à trouver une CFC maximale terminale, la retirer de l'automate, puis itérer à nouveau jusqu'à trouver une CFC maximale terminale acceptante ou un automate vide.

Heureusement il est possible de déterminer toutes les CFC maximales d'un automate en ne parcourant chacune de ses transition qu'une seule fois. L'algorithme le plus connu est dû à Tarjan [115, 116]. Un autre, plus simple mais moins connu, fut développé indépendamment par Dijkstra [42, 43]. Tous deux procèdent à peu près de la même façon: un parcours en profondeur étiquette les états au fur et à mesure qu'ils sont rencontrés, et cela de façon à repérer les CFC. Le second est plus intéressant que le premier dans la mesure où il nécessite de stocker moins d'information par état (un entier par état au lieu de deux pour Tarjan).

Les algorithmes d'*emptiness check* proposés par Couvreur [34], Geldenhuys et Valmari [59, 60] et Hammer et al. [70] fonctionnent sur ce principe.

5.4.1 Algorithme de Couvreur

La figure 5.8 page ci-contre montre une version itérative de l'algorithme de Couvreur [34] sur laquelle nous introduirons deux heuristiques section 5.4.2 page 127.

Avant de détailler cet algorithme, la figure 5.9 page 122 en illustre le principe sur un exemple. L'algorithme explore l'automate par un parcours en profondeur. Chaque fois que l'automate rencontre un nouvel état, il considère cet état comme une CFC triviale. Cette CFC est empilée sur une pile de CFC maintenue parallèlement à (et traversée par) la pile de recherche. Les CFC découvertes par l'algorithme sont représentées sur la figure par des rectangles aux coins arrondis. Après avoir visité successivement les états s_1, s_2, s_3 et s_4, l'algorithme a donc empilé quatre CFC (a). Lorsque l'algorithme visite la transition entre s_4 et s_3, il détecte qu'il s'agit d'une transition entre deux CFC traversées par la pile de recherche: il en déduit que ces deux CFC sont reliées par un circuit et peut donc en faire l'union (b). Lors de cette union, la CFC est étiquetée par toutes les conditions d'acceptation qui appartenaient à ce circuit. Le parcours revient ensuite en arrière (c) ; lorsqu'il quitte cette CFC, on sait qu'elle ne peut contenir de circuit acceptant, et elle peut être supprimée: on marque ces états comme *retirés* (d). Quand par la suite le parcours en profondeur rencontre un état *retiré* (e) il peut l'ignorer. La transition $s_5 \rightarrow s_1$ provoque la fusion des trois CFC sur la pile entre s_1 et s_5 et l'étiquetage par les conditions d'acceptation qui séparaient ces CFC dans la pile de recherche (f). Pour l'instant cette CFC n'est étiquetée que par « ● », qui vient de la transition $s_2 \rightarrow s_5$; il faudra attendre que le parcours en profondeur revienne en arrière (g) puis visite la transition $s_2 \rightarrow s_1$ pour que l'étiquette « ○ » soit ajoutée à la CFC (h). En présence d'une CFC étiquetée par l'ensemble des conditions d'acceptation, l'algorithme peut enfin conclure que $\mathscr{L}(A) \neq \emptyset$.

Dans l'algorithme, *todo* est la pile de recherche associée au parcours en profondeur. Chacun de ses éléments contient un état $state_i$ et l'ensemble $succ_i$ de ses successeurs qui n'ont pas encore été visités. Dans la preuve, nous noterons cette pile

$$todo = \langle state_0, succ_0\rangle\langle state_1, succ_1\rangle \dots \langle state_m, succ_m\rangle$$

$todo[m] = \langle state_m, succ_m\rangle$ désignant le dernier élément ajouté. En pratique, il est souvent possible de représenter ces ensembles de successeurs de façon implicite, par exemple en ne retenant que le dernier successeur visité. On appellera $state_0 \dots state_1 \cdots state_m$ le *chemin de recherche*.

H est un tableau associatif qui indique si un état visité q fait partie d'une CFC maximale qui a été « retirée » de l'automate ($H[q] = 0$) ou si l'état est sur la pile de recherche ($H[q] > 0$), dans ce dernier cas $H[q]$ donne le rang de l'état dans le parcours en profondeur.

Parallèlement à la pile de recherche *todo*, l'algorithme maintient une pile de composantes fortement connexes: SCC. Nous la noterons

$$SCC = \langle root_0, la_0, acc_0, rem_0\rangle\langle root_1, la_1, acc_1, rem_1\rangle \dots \langle root_n, la_n, acc_n, rem_n\rangle$$

avec $SCC[n] = \langle root_n, la_n, acc_n, rem_n\rangle$ désignant la dernière CFC empilée.

À chaque CFC $SCC[i]$ sont associés le rang du premier état de la composante ($root_i$), l'union des conditions d'acceptation présentes dans la CFC (acc_i), les conditions d'acceptation de la transition provenant de la CFC précédente dans le parcours en profondeur

```
1  Entrée : Un TGBA A = ⟨AP, Q, Q^0, F, δ⟩
2  Résultat : ⊤ si et seulement si 𝒳(A) = ∅
3  Données : todo: stack of ⟨state ∈ Q, succ ⊆ δ⟩
             SCC: stack of ⟨root ∈ ℕ, la ⊆ F, acc ⊆ F, rem ⊆ Q⟩
             H: map of Q ↦ ℕ
             max ← 0
4  begin
5      foreach q^0 ∈ Q^0 do
6          DFSpush(∅, q^0)
7          while ¬todo.empty() do
8              if todo.top().succ = ∅ then
9                  DFSpop()
10             else
11                 pick one ⟨s, _, a, d⟩ off todo.top().succ
12                 if d ∉ H then
13                     DFSpush(a, d)
14                 else if H[d] > 0 then
15                     merge(a, H[d])
16                     if SCC.top().acc = F then return ⊥
17     return ⊤
18 end
19 DFSpush(a ⊆ F, q ∈ Q)
20     max ← max + 1
21     H[q] ← max
22     SCC.push(⟨max, a, ∅, ∅⟩)
23     todo.push(⟨q, {⟨s, l, a, d⟩ ∈ δ | s = q}⟩)
24 end
25 DFSpop()
26     ⟨q, _⟩ ← todo.pop()
27     SCC.top().rem.insert(q)
28     if H[q] = SCC.top().root then
29         foreach s ∈ SCC.top().rem do
30             H[s] ← 0
31         SCC.pop()
32 end
33 merge(a ⊆ F, t ∈ ℕ)
34     r ← ∅
35     while t < SCC.top().root do
36         a ← a ∪ SCC.top().acc ∪ SCC.top().la
37         r ← r ∪ SCC.top().rem
38         SCC.pop()
39     SCC.top().acc ← SCC.top().acc ∪ a
40     SCC.top().rem ← SCC.top().rem ∪ r
41 end
```

FIG. 5.8: Autre présentation de l'algorithme de Couvreur [34] pour tester la vacuité d'un TGBA. Cet algorithme ne diffère de l'original que par l'emploi de *rem* et sa présentation itérative (plutôt que récursive).

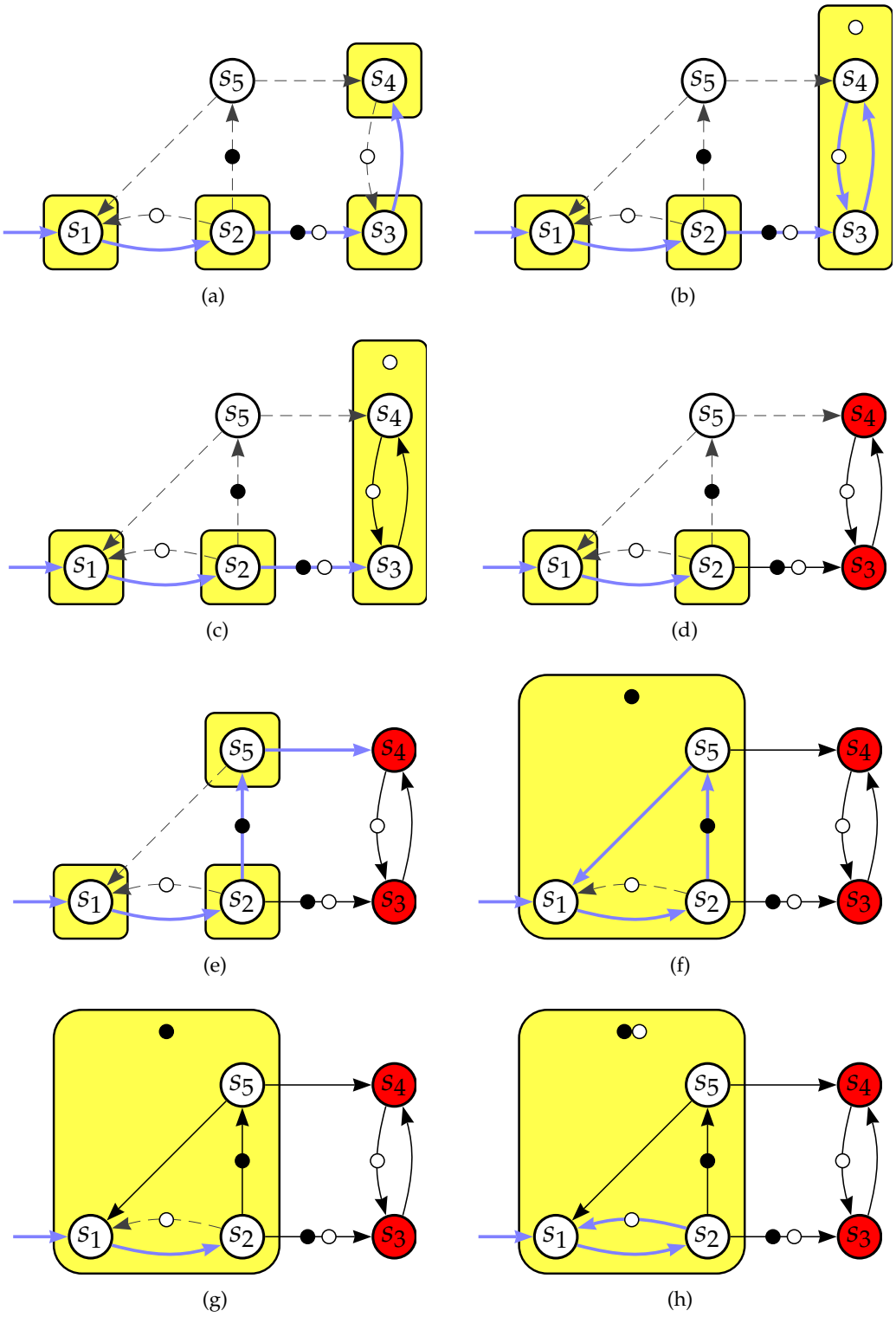


FIG. 5.9: Déroulement de l'algorithme Cou99.

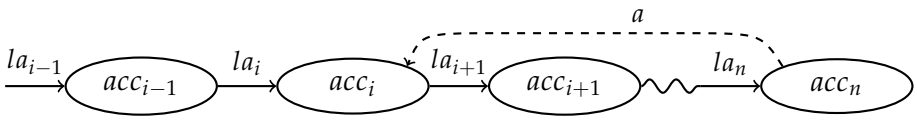


FIG. 5.10: Signification de *acc* et *la* dans *SCC*. Le champ acc_i contient l'ensemble des conditions d'acceptation traversées par les transitions sortantes des états de la CFC $SCC[i]$. la_i indique les conditions d'acceptation portées par la transition reliant $SCC[i-1]$ à $SCC[i]$.

(la_i), et enfin la liste des états de la CFC qui ont été entièrement explorés (rem_i). La figure 5.10 illustre l'emploi de la_i et acc_i.

Pour la toute première composante de la pile, $SCC[0]$, le champ la_0 n'a pas de sens puisqu'il n'y a pas de CFC précédente. Nous l'initialisons à $\emptyset$ ligne 6 mais il pourrait recevoir n'importe quelle valeur: il n'est jamais lu par l'algorithme.

Avec les structures H et *SCC*, nous pouvons dire que deux états q_1 et q_2, non *retirés* (i.e. $H[q_1] \neq 0$ et $H[q_2] \neq 0$) appartiennent à la même CFC si

$$\max\{r \mid SCC[r].root \leq H[q_1]\} = \max\{r \mid SCC[r].root \leq H[q_2]\}$$

et nous pouvons aussi construire l'ensemble $\mathfrak{S}_i$ des états visités appartenant à la CFC représentée par $SCC[i]$:

$$\begin{aligned} \mathfrak{S}_i &= \{s \in \mathcal{Q} \mid root_i \leq H[s] < root_{i+1}\} \qquad \text{pour } 0 \leq i < n \\ \mathfrak{S}_n &= \{s \in \mathcal{Q} \mid root_n \leq H[s]\} \end{aligned} \tag{5.1}$$

Vis-à-vis de cet algorithme d'*emptiness check*, les états de l'automate A peuvent être partitionnés en trois ensembles:
- Les états *actifs* sont ceux qui apparaissent dans H et sont associés à une valeur non nulle,
- les états *retirés* sont ceux qui apparaissent dans H avec une valeur nulle,
- enfin, les états *non explorés* sont ceux qui n'apparaissent pas dans H.

Initialement, tous les états sont *non explorés*. La fonction `DFSpush` est le seul endroit où un état peut passer de l'ensemble des états *non explorés* à celui des états *actifs*. Chaque nouvel état y est empilé comme une CFC triviale avec un ensemble *acc* vide (ligne 22).

Quand le parcours en profondeur atteint un successeur q qui a déjà été visité, mais sans avoir été retiré (ligne 14), c'est-à-dire un état d'une CFC de la pile de recherche, toutes les CFC situées entre la CFC à laquelle q appartient, et les CFC du haut de la pile (à laquelle appartient l'état qui a mené à q) sont fusionnées en une seule CFC (ligne 15). Sur l'exemple de la figure 5.10 où une transition est trouvée entre $SCC[n]$ et $SCC[i]$, les dernières CFC (de i à n) seraient fusionnées en une seule avec les conditions d'acceptation $SCC[i].acc \cup SCC[i+1].la \cup SCC[i+1].acc \cup \cdots \cup SCC[n].la \cup SCC[n].acc \cup a$. Si cette dernière union est $\mathcal{F}$, alors une CFC accessible, non-triviale, et acceptante vient d'être trouvée, et l'algorithme peut retourner $\perp$ (l'automate n'est pas vide) immédiatement (ligne 16).

Au moment où, lors du parcours en profondeur, la racine d'une CFC doit être dépilée (testé ligne 28), on sait que la CFC est maximale, mais non-acceptante. À partir de ce moment ses états peuvent être ignorés lorsqu'ils sont rencontrés à nouveau. Pour cela, l'ensemble des états de la CFC sont marqués comme *retirés* (avec $H[q] = 0$). La fonction `DFSpop` est l'unique endroit où un état peut passer de l'ensemble des états *actifs* à celui des états *retirés*.

Pour prouver la correction de cet algorithme, nous montrons que les invariants suivants sont préservés par toutes les lignes de la fonction principale (lignes 4–17):

Proposition 12. *$m \geq n$ (dans les notations de todo et SCC ci-dessus) et il existe une fonction strictement croissante f telle que $\forall i \leq n, root_i = H[state_{f(i)}]$. Autrement dit, $root_0 \cdot root_1 \cdots root_n$ est une sous-séquence de $H[state_0] \cdot H[state_1] \cdots H[state_m]$. (Ou encore: les racines des composantes fortement connexes sont sur le chemin de recherche du parcours en profondeur, et dans le même ordre.)*

Proposition 13. *Pour tout entier $i \leq n$, le sous-graphe induit par les états de $\mathfrak{S}_i$ est une CFC. De plus, il existe un circuit dans cette CFC qui visite toutes les conditions d'acceptation de acc_i. Enfin, $\mathfrak{S}_0, \mathfrak{S}_1, \ldots, \mathfrak{S}_n$ est une partition de l'ensemble des états actifs.*

Proposition 14. $\forall i < n, \exists s \in \mathfrak{S}_i, \exists p \in 2^{2^{AP}}, \langle s, p, la_{i+1}, state_{f(i+1)} \rangle \in \delta$.

Autrement dit, il existe une transition étiquetée par la_{i+1} entre les CFC d'indice i et $i+1$.

Proposition 15. *Pour tout entier $i \leq n$, rem_i contient tous les états de $\mathfrak{S}_i$ qui ne sont pas sur le chemin de recherche.*

Proposition 16. *L'ensemble des états retirés est une union de composantes fortement connexes maximales. D'autre part pour tout état* retiré *q,* $\mathrm{Acc}(A[\{q\}]) = \emptyset$ *(c'est-à-dire qu'il n'existe aucun chemin acceptant à partir de q dans l'automate A).*

Les deux premières propositions assurent que si l'algorithme trouve un i tel que $acc_i = \mathcal{F}$, alors $SCC[i]$ représente une composante fortement connexe acceptante (prop. 13) et accessible (prop. 12). La dernière proposition affirme qu'aucun chemin acceptant n'existe si l'ensemble des états a été *retiré*.

Démonstration. Les cinq propositions (12–16) sont vérifiées quand la ligne 7 est atteinte pour la première fois. Il n'y a alors qu'un élément dans *todo* et *SCC*, et la façon dont il y a été empilé par `DFSpush` conserve la propriété 12. $\mathfrak{S}_0$ contient seulement un état, donc la propriété 13 est vérifiée. Comme $n = m = 0$, les propriétés 14 et 15 sont triviales. Aucun état n'a encore été *retiré*, donc la propriété 16 est vraie.

Quand une transition $\langle r, \_, a, d \rangle$ est retirée de $succ_m$, nous nous trouvons dans l'un des trois cas suivants:

– $d \notin H$ (ligne 12). d n'a jamais été exploré. Nous lui donnons donc un rang dans H et le considérons comme une CFC triviale en l'empilant sur *SCC*, et continuons le parcours en profondeur vers ses successeurs en l'empilant sur *todo*. Le rang qui lui est affecté nous assure que les propositions 12–14 sont préservées, sans affecter les propositions 15–16.

– $H[d] > 0$ (ligne 14). d est un état *actif*. Notons $root_i$ la plus grande racine de CFC telle que $root_i < H[d]$ et notons $r_i = state_{f(i)}$ l'état associé. (f étant la fonction définie dans la proposition 12.)
D'après la proposition 13, r_i et d sont dans la même CFC. Comme r_i et s sont sur le chemin de recherche, r_i peut atteindre s. Comme nous sommes en train de considérer une transition entre s et d, on en déduit que r_i, s et d appartiennent à la même CFC.
Nous pouvons donc fusionner toutes les CFC au dessus de $root_i$. La nouvelle CFC est l'union de $\mathfrak{S}_i, CS_{i+1} \ldots, CS_n$, et dans cette CFC il existe un circuit qui traverse les conditions d'acceptation $acc_i, acc_{i+1}, \ldots, acc_n$ de chacune des CFC fusionnées aussi bien que les conditions d'acceptation $la_{i+1}, la_{i+2}, \ldots, la_n$ qui les séparaient, ainsi que les conditions d'acceptation a étiquetant la transition que l'on considère. La figure 5.10 montre la situation avant la fusion.
La fonction `merge` s'occupe de fusionner ces conditions d'acceptation afin de préserver la proposition 13. Elle fusionne aussi les états de *rem* pour satisfaire la proposition 15. Les trois autres propriétés ne sont pas affectées par cette opération.
– $H[d] = 0$. Cela signifie que d est un état retiré. D'après la propriété 16, ni d ni aucun de ses descendants ne peut faire partie d'un chemin acceptant. Cet état est donc ignoré par l'algorithme. Comme aucune structure de données n'est altérée, les propositions 12–16 sont préservées.

Après chaque appel de `merge`, c'est-à-dire chaque fois que l'on a créé une CFC non-triviale, l'algorithme vérifie (ligne 16) si cette CFC n'est pas étiquetée par $\mathcal{F}$. Dans ce cas, l'algorithme peut répondre immédiatement par la négative: l'automate n'est pas vide car il existe un circuit acceptant dans cette CFC qui est accessible depuis l'état initial.

Nous nous tournons maintenant vers le cas $succ_m = \emptyset$, testé ligne 8. Les propriétés du parcours en profondeur impliquent alors que tous les successeurs de $state_m$ ont été explorés, ainsi que leur descendants.

L'état $state_m$ est donc retiré du chemin de recherche (ligne 26) et pour préserver la proposition 15 il est donc ajouté à rem_n (ligne 27). Retirer $state_m$ du chemin de recherche n'affecte pas la proposition 12 tant que celui-ci n'est pas la racine d'une CFC. S'il l'est, la CFC doit être retirée aussi. Cette opération n'affecte pas la proposition 14, mais pour préserver la proposition 13 nous devons alors *retirer* les états $\mathfrak{S}_i$ de l'ensemble des états *actifs*. À cet endroit, nous savons que la CFC du haut de la pile est maximale, en effet:

– aucun état *non exploré* ne peut faire partie de cette CFC puisque nous avons visité tous les descendants de $state_m$,
– aucun état *actif* d'une CFC inférieure ne peut faire partie de cette CFC: l'un des descendant de $state_m$ aurait alors causé la fusion de ces CFC,
– aucun état *retiré* ne peut faire partie de cette CFC puisque la propriété 16 nous dit que les états retirés forment des CFC maximales.

Nous savons aussi que cette CFC maximale ne contient pas de circuit acceptant, autrement l'algorithme aurait terminé ligne 16 après avoir traité la dernière transition de ce circuit. Par conséquent $\forall q \in \mathfrak{S}_i, \mathrm{Acc}(A[q]) = \emptyset$, et nous pouvons *retirer* tous ces états sans invalider la proposition 16. Grâce à la proposition 15, à la ligne 29, rem_n contient tous les états de $\mathfrak{S}_i$, donc cette boucle retire bien tous les états de la CFC comme le fait la proposition 13.

Si l'algorithme termine ligne 17, la pile de recherche *todo* est vide. D'après la propriété 12, SCC est aussi vide, et d'après la propriété 13 cela signifie qu'il n'existe plus d'état actif. Comme le DFS a exploré tous les états accessibles de l'automate, nous en concluons que tous les états accessibles ont été retirés. Finalement, la propriété 16 implique que $\mathrm{Acc}(A) = \emptyset$.

Nous avons montré que si l'algorithme termine ligne 16, alors il existe un chemin acceptant, et que s'il termine ligne 17 il n'y en a pas. La terminaison de l'algorithme est assurée par le fait qu'il s'agit d'un parcours en profondeur d'un automate fini.

En conclusion, cet algorithme retourne $\bot$ si et seulement si l'automate contient un chemin acceptant. □

Discussion. L'emploi de *rem* ligne 29 pour retirer les état de la CFC pourrait être évité car, lorsque la ligne 29 est atteinte, *rem* contient tous les états accessibles depuis q (en ignorant ceux qui sont déjà marqués avec $H[q] = 0$). Il est donc possible de retrouver *rem* en faisant un parcours en profondeur depuis q ; c'est ce que faisait l'algorithme original [34] et cette seconde visite de chaque transition a été critiquée par Schwoon et Esparza [107]. L'implémentation proposée ici prend le parti de favoriser le temps d'exécution aux dépends de la consommation mémoire, en ne parcourant ainsi chaque transition qu'une seule fois. Nous tiendrons compte de cet encombrement supplémentaire lors de nos mesures.

Geldenhuys et Valmari [60] ont proposé des structures de données différentes pour résoudre le même problème (visiter chaque transition une seule fois) en combinant l'algorithme original de Couvreur avec les idées d'un algorithme [59] qu'ils avaient proposé avant de découvrir celui de Couvreur. L'algorithme de Geldenhuys et Valmari [59] fonctionne semblablement: il conserve tous les états des CFC partielles afin de pouvoir les retirer aisément. Cependant cet algorithme souffre de deux handicaps: d'une part il est basé sur l'algorithme de Tarjan et a donc besoin d'un entier de plus par état, d'autre part cet algorithme ne fonctionne que sur des automates dégénéralisés. La combinaison de leur algorithme avec celui de Couvreur, publiée l'année suivante [60], lève ces deux limitations.

Dans l'algorithme tel qu'il est présenté figure 5.8, on peut remarquer que tout état actif se trouve représenté dans SCC, soit par un $root_i$, soit dans un des ensembles rem_i. En remplaçant $root_i$ par l'état lui-même, il n'est plus nécessaire de numéroter les états actifs pour retrouver à quelle composante ils appartiennent lors de la fusion: il suffit de fusionner la pile jusqu'à trouver un $root_i$ égal à l'état en question, ou un rem_i qui le contient. Ce constat a permis à Li et al. [91] de diminuer la taille de H. Seuls deux bits par état sont nécessaires pour distinguer les trois cas possibles: *non-exploré*, *actif*, *retiré*.

Merz et Sezgin [95] puis Hammer et al. [70] ont proposé un algorithme d'*emptiness check* pour automates alternants faibles (tels que ceux présentés section 4.1.4 page 80). Leur algorithme convertit l'automate alternant en un automate de Büchi généralisé à la volée pendant l'*emptiness check*. Leur traduction d'automate pourrait être couplée avec n'importe quel autre algorithme d'*emptiness check* discuté ici. La vraie partie de leur algorithme d'*emptiness check* suit la même logique que celui de Couvreur à une différence près: il fusionne les SCC une par une au moment où la CFC est dépilée plutôt qu'immé-

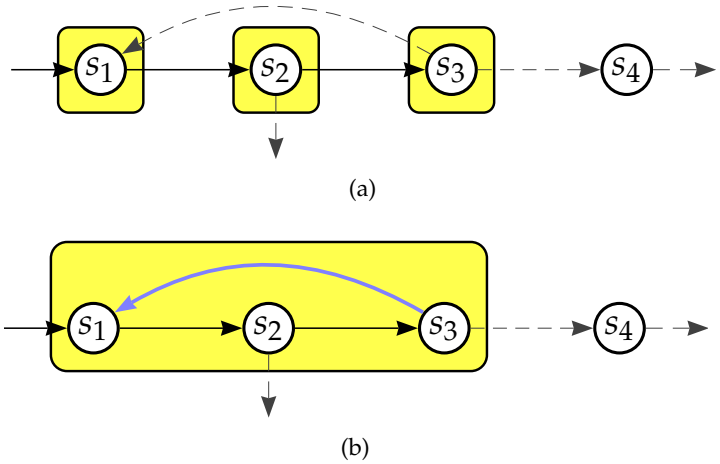


FIG. 5.11: Heuristiques pour l'algorithme Cou99.

diatement lorsqu'un circuit est trouvé. Leur algorithme détectera donc une CFC acceptante plus tard que celui de la figure 5.8 page 121.

5.4.2 Heuristiques

Nous présentons maintenant deux heuristiques qui modifient le comportement de l'algorithme Cou99.

La première s'appuie sur le fait que la ligne 11 de la figure 5.8 page 121 n'impose aucun ordre sur les successeurs. Cou99 utilisera simplement l'ordre naturel dans lequel les successeurs sont générés par l'outil utilisé pour construire l'automate à tester. Une idée (due à Soheib Baarir) serait de visiter en priorité les successeurs déjà vus, puisque soit ce sont des états *retirés* que nous pouvons ignorer, soit ils appartiennent à une CFC et permettront à l'algorithme de faire des fusions. Ainsi, sur l'exemple de la figure 5.11a, l'algorithme, actuellement dans l'état s_3, devrait choisir de visiter s_1 de préférence à s_4. Du point de vue de l'algorithme on peut dire que cette heuristique tente de favoriser les `merges` par rapport aux `pushs`. Nous appellerons Cou99 shy- l'algorithme implémentant cette heuristique. Ce réordonnancement des successeurs n'est pas gratuit car pour chaque état l'algorithme doit

– connaître l'ensemble des successeurs pour faire son choix (cela nuit aux implémentations qui souhaitent calculer les successeurs un par un à la demande)
– garder trace des successeurs qu'il n'a pas encore visités de façon explicite (plutôt que d'utiliser un itérateur)

Ces différences apparaîtront dans nos mesures: ces deux variantes visiteront plus de transitions (parce que tous les successeurs doivent être calculés même s'ils ne sont pas forcément visités) et utiliseront plus de pile (pour stocker les successeurs qui restent à visiter).

La seconde heuristique, que nous nommerons Cou99 shy fonctionne de la même manière mais en considérant les successeurs du point de vue de la CFC entière et non uniquement ceux de l'état en haut de *todo*. Par exemple dans la figure 5.11b, la prochaine transition

franchie par l'algorithme sera soit $s_3 \rightarrow s_4$, soit celle descendant de s_2. Un tel choix laisse plus de liberté lors de la recherche de successeurs déjà vus. Dans cette variante, les deux piles *SCC* et *todo* peuvent être combinées et les successeurs en suspens de toutes les CFC seront réunis lors des fusions.

La plupart du temps, ces heuristiques permettront en effet de visiter moins d'états. Cependant, comme cela vient au prix d'un calcul de plus de transitions et d'une consommation mémoire plus importante, il est difficile de tirer des conclusions définitives. (Geldenhuys et Valmari [59] ont aussi testé différentes heuristiques et observé, sans plus de succès, qu'il n'y en avait aucune qui apparaisse strictement meilleure qu'une autre.) Sur des automates non-vides, on peut même trouver des cas (par exemple en bas à gauche du tableau 5.4 page 135) où ces variantes visitent plus d'états. L'un des problèmes de l'évaluation d'algorithmes d'*emptiness check* à la volée est que l'algorithme va s'arrêter dès qu'il peut: un algorithme complexe peut s'arrêter avant un algorithme plus efficace s'il a la chance de prendre des successeurs dans un ordre qui l'amène rapidement à un circuit acceptant. À part Cou99 shy et Cou99 shy-, qui réordonnent les successeurs, tous les autres algorithmes mesurés ici visitent les successeurs dans le même ordre dans les parcours en profondeur: cela permet des mesures plus équitables.

5.5 Étude de performances

Dans cette section nous présentons un cadre de comparaison expérimentale des algorithmes d'*emptiness check* susmentionnés et critiquons leurs résultats. Tous les algorithmes qui sont mesurés ici ont été implémentés dans la bibliothèque Spot (annexe A page 201), ce qui nous permet de les tester de façon homogène, sur des problèmes identiques. Cet ensemble de tests est lui aussi distribué avec Spot, et a déjà été utilisé par d'autres personnes (Tauriainen [123] s'en est servi pour évaluer une nouvelle heuristique).

Pour certains tests, nous utilisons des graphes et des formules LTL générés aléatoirement. Les algorithmes utilisés pour cette génération sont inspirés de ceux présentés par Tauriainen [117]. Le générateur de graphes est paramétré par un nombre n de nœuds à connecter dans le graphe, une densité d (probabilité de connexion entre deux nœuds), un nombre n_{acc} de conditions d'acceptation, une probabilité a d'appartenir à un ensemble d'acceptation, une liste de propositions atomiques et une probabilité t qu'une propriété atomique soit vraie ou fausse pour une transition. L'algorithme assure que chaque nœud est connecté à un nombre de successeurs compris entre 1 et n suivant une distribution normale de moyenne $1 + (n-1)d$ et de variance $(n-1)d(1-d)$[3]. Tous les nœuds sont ainsi connectés lorsque $d = 1$ alors que chaque nœud n'a qu'un successeur si $d = 0$. Les transitions ainsi créées sont étiquetées par des conditions d'acceptation et propriétés atomiques en fonction de a et t.

L'algorithme de génération d'une formule LTL aléatoire a déjà été décrit page 95. Ici, chaque formule ainsi générée est ensuite simplifiée par les algorithmes présentés sec-

[3]Pour chaque nœud, cela revient à le connecter à un autre nœud pris au hasard, puis à décider de la connexion avec les $n-1$ autres (lui-même inclus) selon une probabilité d. L'emploi d'une loi normale pour décider du nombre de successeurs est une approximation qui permet d'éviter $n-1$ tirages aléatoires coûteux.

tion 4.3.3 page 94 et ne sera conservée que si sa simplification n'a pas changé de taille. Ce filtrage évite qu'une formule comme $\neg\neg\neg\neg p$ soit produite comme une formule de taille 5.

Des huit algorithmes d'*emptiness check* que nous comparerons, les quatre premiers sont basés sur les CFC: Cou99 est l'algorithme de la figure 5.8 page 121, Cou99 shy- et Cou99 shy correspondent aux deux heuristiques décrites section 5.4.2 page 127 et GV04 est l'algorithme de Geldenhuys et Valmari [59]. Les quatre autres sont des NDFS: CVWY90 est l'« ancêtre » des algorithmes non généralisés [32], SE05 est sa dernière évolution par Schwoon et Esparza [107], Tau03 est la première version sur les transitions de l'algorithme de Tauriainen [120] et Tau03 opt est la variante de Tau03 présentée section 5.3.3. Les $\times$ en marge des tableaux indiquent les nouveaux algorithmes introduits dans ce chapitre.

Comme tous nos tests utilisent des TGBA en entrée de l'*emptiness check*, les algorithmes CVWY90, GV04 et SE05 ont du être ajustés. Les modifications qui leur permettent de travailler sur les transitions et non les états sont mineures, en revanche comme ces algorithmes ne supportent pas les conditions d'acceptation généralisées, nous devons dégénéraliser les automates sur lesquels ils doivent travailler, avec l'algorithme de la section 3.3.6 page 55 (l'automate à tester possède n conditions d'acceptation, l'algorithme devra donc travailler sur un automate jusqu'à n fois plus gros).

Nous faisons tourner ces algorithmes sur les graphes aléatoires aussi bien que sur des graphes représentant des modèles concrets, dans une approche similaire à celle de Geldenhuys et Valmari [59].

Le tableau 5.2 page suivante présente les résultats obtenus lors de la vérification de graphes aléatoires avec tous les algorithmes, dans 12 configurations différentes. Pour chaque configuration, les valeurs présentées sont des moyennes obtenues en exécutant un algorithme d'*emptiness check* sur 1300 à 3000 automates résultant chacun du produit d'un graphe aléatoire (représentant un espace d'état) avec un TGBA traduisant une formule LTL. Les configurations diffèrent dans la façon dont le graphe et les formules sont générés.

Chaque graphe aléatoire possède $n = 1024$ états. Ces états sont tous accessibles et leur nombre de successeurs suit une distribution normale de moyenne $1 + 1023d$ et de variance $1023d(1-d)$. Les différentes valeurs de d (0.001, 0.002 et 0.01, correspondant respectivement à une moyenne de 2, 3 et 11 successeurs) sont indiquées sur la gauche du tableau et séparent les 12 configurations en trois rangées utilisant des densités égales. Intuitivement, plus un graphe est dense, plus il a de chances de contenir des circuits acceptants.

Dans les colonnes repérées par *cond. acc. supp.*, les transitions du graphe sont étiquetées par 3 conditions d'acceptation distribuées de façon aléatoire ($a = 1/75$: chaque condition a une chance sur 75 d'apparaître sur une transition). Sans rentrer dans les détails, on peut considérer que l'ajout de ces conditions d'acceptation supplémentaires simule trois hypothèses d'équité faibles sur le modèle que représenterait le graphe aléatoire. Nous reviendrons sur les hypothèses d'équité dans le chapitre 7. Ces trois conditions d'acceptation, malgré leur faible nombre, suffisent à montrer que certains algorithmes seront sérieusement handicapés par ces hypothèses d'équité.

		Formules aléatoires						Formules prédéfinies						
	Algorithme	cond. acc. LTL			cond. acc. supp.			cond. acc. LTL			cond. acc. supp.			
	$d = 0.001$	*2328 (1318)*			*2188*			*2308 (2127)*			*1951*			
	Cou99	6.8	4.5	4.5	18.1	11.1	13.8	7.4	5.1	4.0	16.3	10.6	10.4	CFC
×	Cou99 shy-	5.4	5.8	7.5	16.5	17.7	25.6	6.5	6.8	7.7	15.2	15.7	19.6	CFC
×	Cou99 shy	5.4	5.8	7.4	15.6	16.7	23.5	6.3	6.5	7.0	14.5	15.0	18.0	CFC
dégénéralisés	GV04	6.8	4.5	4.5	28.4	17.1	21.6	7.5	5.1	4.0	25.9	16.5	16.1	CFC
dégénéralisés	CVWY90	6.8	7.1	6.4	61.7	73.9	66.6	7.7	7.9	5.2	53.6	65.0	49.3	NDFS
dégénéralisés	SE05	6.8	5.7	4.5	59.4	39.1	38.4	7.6	6.8	3.9	50.9	34.7	28.1	NDFS
	Tau03	9.9	16.1	10.8	64.7	295.9	49.6	9.5	17.4	8.1	53.9	265.5	36.2	NDFS
×	Tau03 opt	6.8	5.2	4.5	18.5	27.1	15.4	7.4	8.1	3.8	16.4	31.8	11.3	NDFS
	$d = 0.002$	*2716 (1488)*			*2695*			*2569 (2304)*			*2548*			
	Cou99	4.8	2.2	3.1	17.5	8.5	11.4	4.5	2.6	2.4	13.7	7.4	7.6	
×	Cou99 shy-	3.4	3.4	6.9	15.4	15.8	30.1	3.6	3.8	6.1	12.5	12.1	19.8	
×	Cou99 shy	3.4	3.4	6.8	14.3	14.6	26.5	3.4	3.6	5.4	11.9	11.4	17.3	
	GV04	4.8	2.2	3.1	29.1	14.1	18.5	4.6	2.8	2.4	23.2	12.8	11.9	
	CVWY90	4.9	3.6	4.5	60.3	58.0	59.5	4.8	4.2	3.4	45.1	43.9	35.6	
	SE05	4.8	2.8	3.1	56.8	30.2	32.9	4.7	3.5	2.4	42.0	24.3	19.8	
	Tau03	8.5	12.5	9.1	61.3	265.5	46.5	7.1	12.5	6.3	40.9	185.4	27.3	
×	Tau03 opt	4.8	2.7	3.1	17.8	23.9	12.7	4.5	4.5	2.4	13.9	26.3	8.3	
	$d = 0.01$	*2978 (1569)*			*2979*			*2766 (2441)*			*2765*			
	Cou99	3.5	0.7	2.4	12.3	1.9	8.1	2.6	0.7	1.4	7.8	1.5	4.8	
×	Cou99 shy-	1.7	1.5	11.7	8.2	8.3	66.6	1.6	1.6	10.6	5.6	5.4	39.2	
×	Cou99 shy	1.6	1.5	10.7	6.9	7.0	53.0	1.4	1.3	7.7	4.8	4.4	29.9	
	GV04	3.5	0.7	2.4	20.7	3.5	13.2	2.6	0.8	1.4	13.5	2.8	7.7	
	CVWY90	3.6	1.1	3.3	44.7	15.6	49.8	2.7	1.0	2.1	30.8	13.0	30.9	
	SE05	3.6	0.9	2.3	39.8	7.0	25.4	2.6	0.9	1.5	26.4	5.7	15.5	
	Tau03	16.9	20.6	19.7	58.1	221.8	57.8	11.2	14.9	12.7	35.5	140.2	32.3	
×	Tau03 opt	3.5	1.0	2.3	12.4	11.0	9.6	2.6	1.6	1.4	7.8	10.2	5.7	

TAB. 5.2: Comparaison des algorithms d'*emptiness check* sur des graphes aléatoires synchronisés avec des formules aléatoires ou réelles.

Dans les colonnes repérées par *cond. acc. LTL* aucune condition d'acceptation n'est ajoutée au graphe généré aléatoirement, les seules conditions d'acceptation apparaissant dans l'automate à tester par l'algorithme d'*emptiness check* sont celles qui ont été amenées par la traduction de la formule LTL en TGBA.

Dans chaque configuration, on génère 15 espaces d'état aléatoires sous la forme de TGBA. Dans les colonnes titrées *formules aléatoires*, chacun des espaces d'état est synchronisé avec 200 TGBA représentant des formules LTL différentes (de taille 5), produisant ainsi 3000 automates à tester. Dans les colonnes titrées *formules prédéfinies*, chacun des espaces d'état est synchronisé avec une liste de 94 formules (ainsi que leurs négations) choisies dans la littérature [45, 49, 112], produisant ainsi 2820 automates à tester.

Dans ce premier test, nous ignorons tous les automates ainsi construits qui ne contiennent aucun chemin acceptant. Ceci nous permet de nous concentrer uniquement sur les automates dans lesquels un algorithme peut répondre sans parcourir l'automate entier: il peut répondre dès qu'il a la preuve qu'un chemin acceptant existe. Nous pouvons ainsi voir quels algorithmes répondent plus vite. Au dessus de chaque configuration, le nombre en italique indique le nombre d'automates non-vides restants et sur lesquels ont été appliqués les algorithmes. Tau03 est un cas particulier pour lequel nous avons dû aussi ignorer les automates sans aucune condition d'acceptation car il ne sait pas les gérer ; le nombre d'automates non-vides possédant au moins une condition d'acceptation est donc indiqué entre parenthèses. (La distinction n'est pas faite dans les colonnes *cond. acc. supp.* puisque les automates y ont toujours au moins 3 conditions.) On peut constater que le nombre d'automates non-vides augmente avec la densité d de l'automate, comme on s'y attendait.

Pour chaque automate non-vide, nous calculons trois valeurs, présentées en ligne et dans cet ordre:

1. le rapport entre le nombre d'états distincts visités par l'algorithme, et le nombre d'états de l'automate (cette valeur peut dépasser 100% dans les cas de CVWY90, GV04 et SE05 où l'automate doit être dégénéralisé afin de pouvoir être exploré par l'algorithme),
2. le rapport entre le nombre de transitions traversées et le nombre de transitions du produit (une même transition traversée plusieurs fois sera comptée autant de fois, c'est là une autre raison de dépasser 100%),
3. le rapport entre la taille maximale de la pile de recherche et le nombre d'états de l'automate

Dans tous les cas, même lorsque l'algorithme requiert une dégénéralisation, ces rapports sont calculés par rapport à l'automate avant sa dégénéralisation. Les valeurs sont affichées sous forme de pourcentages.

La première valeur donne une idée de la portion de l'automate que l'algorithme a dû couvrir avant de pouvoir répondre. La seconde donne une idée du temps passé par l'algorithme à visiter l'automate (on considère la traversée d'une transition comme l'opération clef de l'*emptiness check*, un peu comme on compterait les échanges dans un algorithme de tri). La dernière valeur, combinée à la première, donne une idée de la consommation mémoire de l'algorithme.

En effet, chaque algorithme occupe de la mémoire avec deux types de structures. Une première est une table de hachage qui lui permet d'associer des informations à chaque état visité: cela peut aller d'un bit pour indiquer que l'état a été visité à un entier pour numéroter l'état. On peut considérer que la mémoire occupée par la table de hachage est proportionnelle au nombre d'états visités. On peut même ignorer cette table car la taille des données supplémentaires pour chaque état est négligeable devant la taille de l'état lui-même. L'autre source de consommation mémoire de ces algorithmes réside dans les piles utilisées lors de la recherche. C'est ce qu'essaye de mesurer la dernière valeur. Son calcul demande de plus amples explications. Pour les NDFS, nous comptons simplement le nombre d'états dans la pile de recherche. Pour Cou99, nous comptons le nombre d'entrées de *todo* (la pile de recherche) et y ajoutons tous les états qui doivent être conservés dans les champs *rem* de SCC (*succ* peut être représenté par un itérateur de taille constante, et si l'on omet *rem*, la taille de *SCC* est bornée par celle de *todo*). Pour Cou99 shy et Cou99 shy- *succ* ne pouvant plus être représenté par un itérateur, nous ajoutons chacun de ses éléments à la taille. Pour GV04, nous comptons tous les états de *stack* [59] (il est proportionnel au nombre d'états qui sont dans la chaîne de CFC courante).

Pour résumer, les algorithmes présentés ici ont un temps d'exécution proportionnel au nombre de transitions explorées (seconde valeur). Et pour un nombre de conditions d'acceptation $|\mathcal{F}|$ fixé, la consommation mémoire des algorithmes est une combinaison linéaire du nombre d'états visités (première valeur) et de la taille de la pile (troisième) valeur.

Une première remarque concerne les résultats présentés par Geldenhuys et Valmari [59], qui comparent GV04 et CVWY90 à l'aide d'un tableau similaire (sans les colonnes *cond. acc. supp.*). Nous n'avons pas pu reproduire la différence importante qu'ils ont notée entre ces deux algorithmes. (Les résultats de Hammer et al. [70] ne la montrent pas non plus.) Prenons par exemple le cas où les 94 formules LTL prédéfinies, ainsi que leurs négations, ont été synchronisées avec des graphes aléatoires de densité $d = 0.001$. Sur les 2820 automates produits, 2308 sont non-vides. Sur ces derniers, GV04 signale l'existence d'un chemin acceptant après avoir visité 7.5% des états en moyenne, tandis que CVWY90 en visite 7.7%. De leur côté, Geldenhuys et Valmari [59] obtiennent un taux de 8.99% pour GV04 contre 40.21% pour CVWY90. Ces différences sont peut-être dues à une façon différente de générer les automates aléatoires.

Les colonnes des formules générées aléatoirement et celles des formules prédéfinies présentent des résultats similaires ; cela montre que l'emploi de formules aléatoires aurait tout à fait suffi à évaluer les algorithmes, et que leur génération ne semble pas biaisée. De façon similaire, il y a peu à tirer de l'effet des différentes densités d sur chaque algorithme: plus le graphe est dense, plus il contient de circuits acceptants, et plus les algorithmes répondent vite (le cas de Tau03 est une exception sur laquelle nous reviendrons). Il est plus intéressant de comparer les comportements de ces algorithmes selon que des conditions d'acceptation ont été ajoutées au graphe ou non. Lorsque les seules conditions d'acceptation proviennent de la traduction de la formule LTL, elles sont assez peu nombreuses: sur la liste de formules prédéfinies, 40% demandent 0 ou 1 condition d'acceptation, 40% en utilisent 2 et 20% en ont entre 3 et 6. Par conséquent, les différences entre les algorithmes CVWY90, GV04, SE05 (qui demandent des automates dégénéralisés) et les autres ne sont pas très marquées. Dès que l'on ajoute des conditions d'acceptation, les algorithmes sup-

portant les automates généralisés prennent nettement le dessus, et parmi ceux-ci, ce sont ceux basés sur les CFC qui s'en sortent le mieux.

Les mauvais résultats de Tau03 sont dus à une petite erreur de logique lorsque Tauriainen a adapté son algorithme sur les états [119] en un algorithme sur les transitions [120]. Tel qu'il est présenté, l'algorithme visite forcément tous les successeurs d'un état même s'il aurait pu répondre après avoir visité le premier. Ce détail est corrigé dans l'algorithme présenté figure 5.6 page 116 et a été repris par la suite par Tauriainen [122, 123].

Comme il est possible que ces expérimentations, basées sur des graphes générés aléatoirement, soient éloignées des cas concrets de vérification, nous avons aussi exécuté ces algorithmes sur de vrais modèles. Nous avons à cette fin réutilisé les modèles présentés par Geldenhuys et Valmari [59], modélisant un algorithme d'élection dans un réseau (ce modèle est aussi utilisé par Schwoon et Esparza [107] dans leurs mesures). Le tableau 5.3 page suivante montre nos résultats pour la seconde des trois variations présentées par [59]. Le modèle a été vérifié par rapport aux 9 formules numérotées de A à I.

Chaque case du tableau correspond à une formule donnée. Au dessus d'une case sont indiqués: le numéro de la formule (A à I) ainsi que la taille de l'automate produit en terme d'états, transitions et conditions d'acceptation. De plus, un symbole indique si le produit est vide (∅) ou si un chemin acceptant existe (↻). Pour chaque algorithme, nous indiquons le nombre d'états distincts visités, le nombre de transitions visitées, ainsi que la taille maximale de la pile. Contrairement au tableau précédent, il ne s'agit pas de ratios puisque nous ne cherchons pas à faire de moyennes. Enfin, aucune mesure n'a été effectuée pour Tau03 sur les TGBA sans condition d'acceptation.

Le modèle est écrit en Promela. Le graphe complet[4] de ses états accessibles a été généré avec Spin [72, 75]. Cet espace d'état a ensuite été transformé sous une forme lisible par Spot pour y être introduit sous la forme d'un TGBA et enfin soumis aux différents *emptiness checks*. Bien que ce ne soit généralement pas le cas, sur cet exemple l'automate dégénéralisé a la même taille que l'automate généralisé. C'est pour cette raison que Cou99 et GV04 sont aussi efficaces l'un que l'autre. L'implémentation d'origine de Cou99 [34] aurait utilisé beaucoup moins de pile en visitant le double de transitions (par exemple pour la formule F les résultats de l'implémentation de Cou99 qui n'utilise pas *rem* sont $\langle 289812, 2451598, 1145\rangle$).

Ces exécutions confirment les conclusions de Schwoon et Esparza [107]. SE05 se comporte toujours mieux que CVWY90 (formules A, B, E, G and H) ; et les algorithmes basés sur les CFC, Cou99 et GV04, sont meilleurs que les NDFS (formules A et H).

Pour conclure ces expérimentations et insister sur l'emploi de conditions d'acceptation multiples, voici des mesures complémentaires pour un modèle simple de client-serveur dans lequel c clients communiquent avec s serveurs au travers d'un canal bidirectionel. Tout client peut envoyer une requête aux serveurs. L'un des serveurs reçoit ensuite la requête (dans l'ordre des arrivées) pour y répondre. La propriété que l'on souhaite vérifier est que si le premier client envoie une requête, il obtiendra une réponse. Cette propriété n'est vraie que s'il y a un seul client ou un seul serveur ; autrement on peut avoir le cas où un serveur reçoit le message du premier client, puis le système continue en faisant

[4] C'est-à-dire sans réduction d'ordre partiel. Les conclusions pour les graphes réduits sont les mêmes, seulement avec de plus petites valeurs.

		A (287922, 1221437, 1) ↻			B (287922, 1222805, 1) ↻			C (47887, 134916, 0) ∅		
	Cou99	365	365	365	365	365	365	47887	134916	115
×	Cou99 shy-	365	1356	1358	365	1356	1358	47887	134916	226
×	Cou99 shy	365	1356	1358	365	1356	1358	47887	134916	226
	GV04	365	365	365	365	365	365	47887	134916	115
	CVWY90	17693	91145	902	448	789	787	47887	269831	115
	SE05	17693	90803	564	448	449	449	47887	269831	115
	Tau03	17702	187964	911	448	1876	787			
×	Tau03 opt	365	365	366	365	365	366	47887	134916	115
		D (289812, 1232783, 1) ∅			E (145400, 413351, 0) ↻			F (289812, 1225799, 1) ∅		
	Cou99	289812	1232783	145172	365	365	365	289812	1225799	145172
×	Cou99 shy-	289812	1232783	145666	365	706	708	289812	1225799	145666
×	Cou99 shy	289812	1232783	145304	365	706	708	289812	1225799	145304
	GV04	289812	1232783	145172	365	365	365	289812	1225799	145172
	CVWY90	289812	1642497	1145	365	703	704	289812	1635513	1145
	SE05	289812	1642497	1145	365	365	366	289812	1635513	1145
	Tau03	289812	2875280	1145				289812	2861312	1145
×	Tau03 opt	289812	1642497	1145	365	365	366	289812	1635513	1145
		G (241808, 687630, 1) ↻			H (728132, 2080615, 4) ↻			I (728132, 2076619, 4) ∅		
	Cou99	557	557	557	145847	413799	145172	728132	2076619	145172
×	Cou99 shy-	557	1087	1089	145847	414229	145303	728132	2076619	145307
×	Cou99 shy	557	1087	1089	145847	414229	145257	728132	2076619	145257
	GV04	557	557	557	145847	413799	145172	728132	2076619	145172
	CVWY90	557	895	896	178543	511930	1388	728132	2489217	1172
	SE05	557	557	558	178543	504468	1145	728132	2489217	1172
	Tau03	566	1249	905	178551	1604336	1454	728132	6631906	1454
×	Tau03 opt	557	557	558	145847	827149	1454	728132	4555287	1454

TAB. 5.3: Algorithme d'élection d'un maître sur un réseau donné.

		3 clients, 1 serveur ∅				3 clients, 1 serveur, équité ∅			
	Cou99	a	783	2371	511	b	783	2371	511
×	Cou99 shy-	a	783	2371	710	b	783	2371	710
×	Cou99 shy	a	783	2371	519	b	783	2371	519
	GV04	a	783	2371	511	b′	2005	6627	550
	CVWY90	a	783	2897	237	b′	2005	7771	251
	SE05	a	783	2897	237	b′	2005	7771	251
	Tau03	a	783	5268	238	b	783	10143	264
×	Tau03 opt	a	783	2897	237	b	783	8200	264
		3 clients, 3 serveurs ↻				3 clients, 3 serveurs, équité ∅			
	Cou99	c	631	839	159	d	21394	85387	11465
×	Cou99 shy-	c	631	1153	487	d	21394	85387	17133
×	Cou99 shy	c	1170	1914	401	d	21394	85387	11469
	GV04	c	631	839	159	d′	77979	339876	11521
	CVWY90	c	631	1513	159	d′	77979	410877	5632
	SE05	c	631	1499	159	d′	77979	410877	5632
	Tau03	c	899	3373	191	d	21394	415551	5099
×	Tau03 opt	c	631	1499	159	d	21394	331587	5060

taille des produits			
ref.	états	trans.	cond.
a	783	2371	1
b	783	2371	5
b′	2005	6627	1
c	21394	85387	1
d	21394	85387	7
d′	77979	339876	1

TAB. 5.4: Algorithme client-serveur.

évoluer tous les autres processus clients et serveurs. Une façon de corriger ce problème est de faire l'hypothèse d'équité suivante : tous les processus avancent infiniment souvent. Comme nous le verrons section 7.2 page 160, cela revient à ajouter des conditions d'acceptation dans l'automate représentant l'espace des états.

Le tableau 5.4 page précédente montre les mesures obtenues. On y voit en effet que la propriété n'est pas satisfaite pour 3 serveurs sans équité. Le point intéressant est que les conditions d'acceptation ajoutées pour prendre en compte l'hypothèse d'équité ne donnent aucun travail supplémentaire à Cou99 alors que le coût s'en ressent pour les autres algorithmes. Cela se voit sur le cas avec 1 serveur (et cela peut être généralisé), mais il faut prendre garde à ne pas comparer les cas avec 3 serveurs car dans un cas l'automate est vide, et dans l'autre il ne l'est pas.

5.6 Calcul de contre-exemples

Lorsque l'un des algorithmes d'*emptiness check* précédents retourne $\perp$ cela signifie que l'automate à tester n'était pas vide et qu'il possède un chemin acceptant. Si l'automate testé est le produit de l'espace d'état d'un système avec un automate représentant la négation d'une propriété à tester, cela signifie qu'il existe un *contre-exemple*: une exécution du système qui invalide la propriété.

Une étape importante de la vérification sera donc de produire un tel contre-exemple.

En général plusieurs contre-exemples existent, et l'on cherche à en extraire un de petite taille (idéalement de taille minimale): il est plus facile de comprendre un problème si la séquence qui le provoque est courte.

Dans les algorithmes NDFS qui travaillent sur des automates non-généralisés, tels que CVWY90 ou SE05, ces contre-exemples peuvent être produits directement par l'*emptiness check*: lorsque l'algorithme s'arrête, la pile de recherche constituée des états passés en argument de `dfs_blue` et `dfs_red` (cf. figure 5.2 page 110) représente les états du contre-exemple. Cette pile n'indique que les états du contre exemple, pas les transitions qui les relient: lorsqu'il existe plusieurs transitions possibles entre deux états, il suffit de choisir une transition acceptante si elle existe, ou n'importe quelle autre transition autrement.

Dans le cas des *emptiness checks* travaillant sur des automates généralisés, la construction du contre exemple n'est plus aussi simple car il faut prendre en compte des conditions d'acceptation multiples.

Pour GV04, Geldenhuys et Valmari [60] ont montré comment utiliser un entier supplémentaire par état sur la pile pour pouvoir produire un contre-exemple.

Dans cette section nous proposons deux techniques pour calculer des contre-exemples à partir des structures de données des *emptiness checks* génériques Cou99 et Tau03 et leurs variantes. Malheureusement pour ces algorithmes, les structures de données construites ne prouvent l'existence d'un contre-exemple que de façon implicite. Il est nécessaire de parcourir à nouveau l'automate pour en extraire un contre-exemple, mais au moins pouvons-nous guider cette recherche grâce aux données des *emptiness check*.

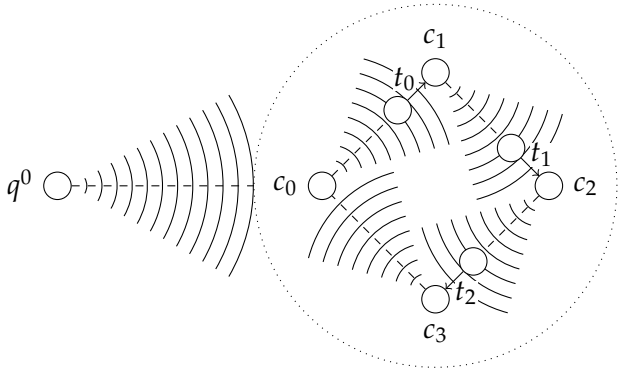


FIG. 5.12: Calcul d'un chemin acceptant dans un TGBA.

Les deux techniques (pour Cou99 et Tau03) fonctionnent sur le même principe: elles essayent de construire un circuit acceptant (qu'on espère petit) à l'aide de plusieurs parcours en largeur, puis construisent le plus petit préfixe amenant à ce circuit.

5.6.1 Contre-exemples pour Cou99

L'algorithme Cou99 retourne $\perp$ lorsqu'il a trouvé une CFC accessible depuis l'état initial q^0. La figure 5.12 représente cette CFC par un cercle en pointillés. On peut facilement savoir si un état s appartient à cette CFC en testant si $H[s] \geq root_n$ (cf. équation 5.1 page 123).

Comme cette CFC est acceptante, on peut trouver, à partir de n'importe lequel de ses états, un circuit étiqueté par toutes les conditions d'acceptation de l'automate. Ce circuit peut devoir franchir une même transition plusieurs fois. Par exemple, dans la CFC de la figure 5.9h page 122, le circuit acceptant $s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1$ passe deux fois par la transition $s_1 \rightarrow s_2$. Par conséquent, il est plus facile de considérer ce circuit comme la concaténation de plusieurs parties qui ne visitent chaque transition qu'une seule fois. Sur l'exemple précédent, l'on pourrait d'abord chercher à construire le circuit $s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_1$ (qui passe par «●»), puis y ajouter le circuit $s_1 \rightarrow s_2 \rightarrow s_1$ (qui passe par «○»). Dans l'algorithme que nous proposons, ces différentes parties ne sont pas forcément des circuits.

Notons $\mathcal{F}_0$ l'ensemble des conditions d'acceptation. À partir d'un état c_0 de la CFC, lançons un parcours en largeur, restreint aux seuls états de la CFC pour construire un chemin vers la plus proche transition t_0 possédant des conditions d'acceptation F_0 telles que $F_0 \cap \mathcal{F}_0 \neq \emptyset$. Notons $CF_1 = \mathcal{F}_0 \setminus F_0$ l'ensemble des conditions d'acceptation restantes, et c_1 la destination de t_1. À partir de c_1, nous lançons un nouveau parcours en largeur restreint à la CFC pour trouver le plus petit chemin qui amène à une transition portant des conditions d'acceptation F_1 dont l'intersection avec $\mathcal{F}_1$ est non-vide. On répète cette procédure jusqu'à ce que $\mathcal{F}_n = \emptyset$. On a alors construit un chemin visitant toutes les conditions d'acceptation. Il ne reste plus qu'à transformer ce chemin en un circuit: on lance un nouveau parcours en largeur depuis c_n pour trouver le plus petit chemin vers c_0, refermant ainsi le circuit.

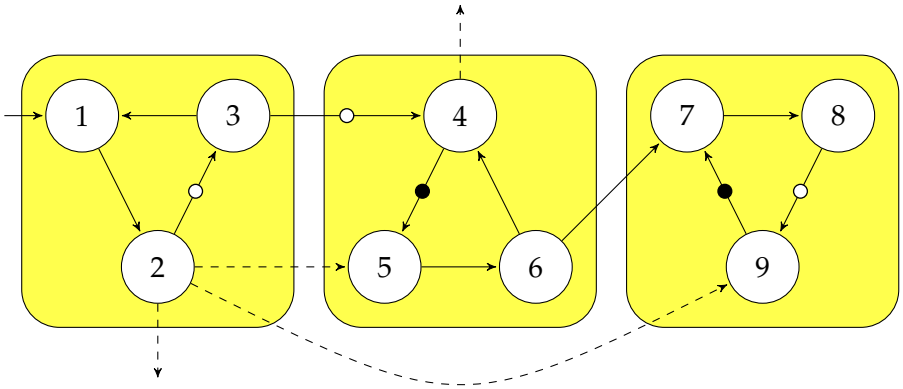


FIG. 5.13: Cas où la génération d'un contre-exemple peut sauter des CFC. $\mathcal{F} = \{\bullet, \circ\}$ donc seule la dernière CFC est acceptante. Les états sont étiquetés par leur valeur dans H après l'exécution de Cou99 shy. Les arcs en pointillés correspondent à des arcs qui n'ont pas été explorés lors du parcours en profondeur pendant l'*emptiness check*, mais pourraient l'être lors du parcours en largeur pour le calcul de contre-exemple. Ainsi l'arc reliant la première CFC à la dernière permet de sauter la seconde.

Cet algorithme a été présenté par Latvala et Heljanko [90] en utilisant la racine de la CFC pour c_0. Ce choix est complètement arbitraire: il se peut que le plus court circuit acceptant ne passe jamais par cet état. Dans le cas de l'algorithme Cou99, nous savons qu'un circuit acceptant n'utilisant que les transitions visitées par l'algorithme, passera forcément par la transition qui a provoqué l'arrêt de l'algorithme (la dernière transition tirée ligne 11 de la figure 5.8 page 121). En effet, s'il avait été possible de construire un chemin acceptant sans franchir cette transition, l'algorithme ce serait arrêté plus tôt. Il semble donc plus sage de débuter la recherche à partir de la source ou de la destination de cette transition: deux états que nous savons appartenir au circuit.[5]

En ce qui concerne le préfixe, une liste d'états reliant un état de $\mathcal{Q}^0$ à c_0 peut-être obtenue facilement à partir de la pile *todo*: il s'agit de ce que nous avons appelé le chemin de recherche page 120. Cependant ce préfixe n'est pas nécessairement le plus court pour atteindre le circuit acceptant: d'une part un autre état du circuit est peut-être accessible plus rapidement, d'autre part il existe peut-être des transitions sortantes encore non explorées au fil de ce parcours et qui le raccourcissent. Pour cette raison nous effectuerons encore un parcours en largeur à partir de q^0 pour tenter d'atteindre n'importe lequel des états du circuit acceptant. Nous limitons la recherche de ce parcours en largeur aux seuls états déjà visités par l'*emptiness check*, et tels que la source d'une transition ait un rang toujours inférieur à la destination ($H[s] \leq H[d]$). Cette dernière force la visite des CFC dans l'ordre croissant, mais en autorisant certaines CFC à être ignorées par des transitions jusque là non-explorées ; la figure 5.13 illustre cette situation.

[5]Il se peut que, malgré ce choix, le circuit construit ne passe pas par cette transition. C'est parce que la recherche n'est pas limitée aux transitions déjà visitées par l'algorithme – il n'en garde pas la trace – mais aux *états* qui constituent cette dernière composante. On peut donc, lors de la recherche du contre-exemple au sein de la dernière CFC, explorer des transitions que l'*emptiness check* n'avait pas encore visitées.

5.6.2 Contre-exemples pour Tau03

Le calcul de chemin acceptant pour un NDFS comme Tau03 ou Tau03 opt est plus délicat parce que leurs données révèlent peu d'informations structurelles aussi utiles qu'une CFC (qui permet de restreindre une recherche).

Nous savons que le dernier état s pour lequel la ligne 23 de la figure 5.6 page 116 a été exécutée appartient au circuit acceptant. Depuis cet état $c_0 = s$ nous lançons un premier parcours en profondeur imbriqué pour trouver un ensemble de transitions $\mathcal{T}$ telles que:
– pour chaque transition t de $\mathcal{T}$ il y existe un circuit passant par c_0 et t,
– l'union des conditions d'acceptation de $\mathcal{T}$ est $\mathcal{F}$.
Cette collection de circuits pourrait être mise bout à bout pour construire un circuit acceptant, mais afin d'essayer de reduire la taille du chemin nous avons décidé de relier ces transitions entre elles sans nécessairement chercher à revenir à c_0.

Nous effectuons donc un parcours en largeur à partir de c_0 pour construire le plus petit chemin atteignant n'importe quelle transitions t_0 de $\mathcal{T}$. À partir de cette transition, nous effectuons un autre parcours en largeur pour atteindre tout autre transitions t_1 de $\mathcal{T}$, etc. La fermeture du circuit et le calcul du préfixe peut se faire comme dans le cas de Cou99, à ceci près que la recherche qui n'est limitée qu'aux états déjà visités par l'*emptiness check* peut être très longue.

5.6.3 Mesures des contre-exemples

Le tableau 5.5 page suivante utilise la même présentation que la tableau 5.2 page 130. Pour chaque configuration, les deux valeurs sont le nombre d'états visités pour construire la partie cyclique du chemin acceptant, et la taille de l'espace de recherche pour ce circuit. Ces valeurs sont exprimées sous la forme d'un pourcentage d'états par rapport à la taille du TGBA testé. Pour Cou99, l'espace de recherche du circuit est la dernière CFC, et pour Tau03 l'espace de recherche est constitué de tous les états présents dans H. Chaque état y est compté autant de fois qu'il est visité.

Ainsi que le montre bien cette table, l'absence d'information structurelle dans Tau03 rend les calculs plus coûteux puisque l'espace de recherche ne peut être réduit autant que Cou99. Dans le cas de Cou99, la restriction de la recherche à un petit sous-graphe (la dernière CFC) justifie l'emploi d'un parcours en largeur. L'ajout de conditions d'acceptation supplémentaires augmente naturellement les parcours de l'espace de recherche.

Le fait que l'espace de recherche de Cou99 shy et Cou99 shy- est plus réduit que celui de Cou99 peut sembler assez surprenant puisque ces heuristiques visent à favoriser les fusions de CFC, mais ces fusions ont justement permis d'interrompre l'algorithme plus tôt en visitant moins d'état.

Lors de nos expériences, nous avons constaté que la taille des chemins acceptants produits avec des parcours en largeur étaient véritablement plus petits que ceux obtenus directement à partir de la pile des parcours en profondeur des algorithmes NDFS. Une étude plus poussée des algorithmes existants, opposant l'importance de la minimisation à la complexité de ce calcul reste à faire. (Gastin et al. [57] donnent quelques pistes.)

	Formules aléatoires				Formules prédéfinies			
Algorithm	cond. acc. LTL		cond. acc. supp.		cond. acc. LTL		conf. acc. supp.	
$d = 0.001$	*2328 (1318)*			*2188*	*2308 (2127)*			*1951*
Cou99	1.9	2.0	17.1	12.7	1.3	1.3	12.3	9.0
Cou99 shy-	1.5	1.5	15.7	11.7	1.0	1.0	11.6	8.4
Cou99 shy	1.5	1.5	15.0	11.0	1.0	1.0	11.0	7.9
Tau03	10.9	9.9	237.6	64.7	8.7	9.5	205.9	53.9
Tau03 opt	2.8	6.8	64.5	18.5	2.2	7.4	53.4	16.4
$d = 0.002$	*2716 (1488)*			*2695*	*2569 (2304)*			*2548*
Cou99	1.3	1.5	15.2	10.6	0.9	1.0	9.7	6.6
Cou99 shy-	0.9	0.9	14.0	9.4	0.6	0.7	8.8	5.9
Cou99 shy	0.9	0.9	13.3	8.6	0.6	0.6	8.5	5.6
Tau03	10.8	8.5	225.1	61.3	7.7	7.1	153.5	40.9
Tau03 opt	2.6	4.8	61.9	17.8	1.8	4.5	43.6	13.9
$d = 0.01$	*2978 (1569)*			*2979*	*2766 (2441)*			*2765*
Cou99	0.8	1.0	12.5	7.4	0.5	0.6	7.1	4.2
Cou99 shy-	0.4	0.4	9.8	4.9	0.2	0.2	5.5	2.9
Cou99 shy	0.4	0.4	9.2	4.1	0.2	0.2	5.1	2.5
Tau03	18.1	16.9	210.0	58.1	12.6	11.2	139.4	35.5
Tau03 opt	1.7	3.5	43.0	12.4	1.1	2.6	25.6	7.8

TAB. 5.5: Comparaison des algorithmes pour le calcul des circuits acceptants.

5.7 *Bit-state hashing*

La technique du *bit-state hashing* [75, page 206] consiste à représenter l'ensemble de l'espace d'état par un tableau de bits de taille fixée $m \ll |\mathcal{Q}|$.

À l'origine présentée dans un contexte d'analyse des états accessibles [71], l'idée a ensuite été généralisée au *model checking*. Les tout premiers algorithmes d'*emptiness check*, des NDFS pour automates de Büchi non généralisés [32, 33], n'ont besoin d'associer qu'un bit par état pour indiquer si celui-ci a été visité. En hachant ces bits sans détection des collisions entre états, l'algorithme peut se tromper et croire que l'automate n'accepte aucune exécution ; en revanche s'il trouve un contre-exemple nous sommes assurés que ce dernier existe vraiment.

Le risque de collisions peut être atténué en utilisant plusieurs fonctions de hachage, soit en indiçant des tableaux différents (multi-hachage), soit en indiçant le même tableau (hachage séquentiel), ou en faisant un mélange des deux (multi-hachage séquentiel). Wolper et Leroy [147] et Holzmann [73] comparent ces différentes méthodes d'un point de vue probabiliste.

Le résultat de l'*emptiness check* doit alors être interprété de façon probabiliste: soit une erreur a été détectée et l'on est sûr que le modèle ne vérifie pas la propriété, soit aucune erreur n'a été détectée et l'on n'est sûr qu'à un certain degré (par exemple 99%) qu'aucune

erreur n'a pu être manquée. Une telle technique relève donc plus de l'aide au débuggage rapide, que de la vérification pure.

Dans tous les cas, même si m est plus petit que $|S|$, il doit être choisi suffisamment grand pour limiter le nombre de collisions. Wolper et Leroy [147] puis Stern et Dill [113] ont montré que ce tableau de m bits peut être stocké de façon beaucoup plus compacte en hachant les indices des bits à 1 dans une table de hachage plus petite avec gestion des collisions. Cette dernière approche est appelée *hash compaction*.

La compatibilité avec la technique *bit-state hashing* est souvent citée comme une raison de privilégier une approche du *model checking* basée sur un NDFS et des automates de Büchi non généralisés.

Nous souhaitons ici indiquer les modifications à apporter pour utiliser cette technique avec les algorithmes d'*emptiness check* que nous avons présentés dans ce chapitre. Tous demandent en effet de s'assurer que lorsqu'un circuit acceptant est détecté par l'algorithme, il correspond réellement à un circuit acceptant.

– Pour l'algorithme SE05 de la figure 5.2 page 110, le circuit acceptant est détecté lorsque l'un des parcours en profondeur atteint un état de la pile de recherche du premier parcours, c'est-à-dire un état cyan. Si le tableau associatif H ne gère plus les collisions il est possible qu'un état qui ne soit pas sur la pile de recherche apparaisse comme cyan parce qu'il partage sa clef avec un état de la pile.

  Ce problème peut être évité en s'assurant que la couleur des états de la pile de recherche du premier parcours en profondeur est stockée de façon sûre: par exemple en enregistrant ces états dans une table de hachage séparée avec gestion des collisions. La consommation de cette table reste bornée par la profondeur maximale de la pile de recherche. Comme les états de ce parcours doivent de toute façon être empilés, la complexité spatiale reste identique.

– Lorsque des poids sont ajoutés à l'algorithme comme dans la figure 5.4 page 113, cette table de hachage séparée peut être fusionnée avec la table représentant les poids. En effet, les poids ne sont stockés que pour les états de la pile de recherche du premier parcours en profondeur.

– L'algorithme Tau03 opt de la figure 5.6 page 116 demande lui aussi à ce que la couleur des états de la première pile de recherche soit préservée. À nouveau, lorsque les poids sont utilisés il y a avantage à utiliser une table de hachage commune.
  Là encore, cette modification demande un espace mémoire proportionnel à la pile de recherche du premier parcours en profondeur.

– Enfin dans l'algorithme Cou99 de la figure 5.8 page 121 un circuit (potentiellement acceptant) est trouvé lorsque l'algorithme découvre un état *qui appartient* à une composante fortement connexe de la pile de recherche. Pour s'assurer du fait qu'un circuit soit acceptant, il faut être sûr de la CFC dans laquelle arrive la transition qui ferme le circuit. Cela n'est possible que si le rang de tous les états *actifs* est préservé. Autrement dit, le *bit-state hashing* ne peut être appliqué qu'aux états *retirés* (les états q tels que $H[q] = 0$).

  Naturellement, ceci limite assez sérieusement l'intérêt du *bit-state hashing* avec cet algorithme.

Dans un article intitulé « *Combining Couvreur's Algorithm with Bitstate-hashing for Emptiness Check* », Li et al. [91] montrent comment le rang associé à chaque état dans H peut être remplacé par deux bits, mais sans jamais expliquer en quoi cela rendrait l'algorithme compatible avec le *bit-state hashing*.

5.8 Conclusion

Les contributions de ce chapitre sont multiples.

Nous avons présenté de façon assez détaillée puis comparé les différents algorithmes d'*emptiness check* pour automates de Büchi existants.

Nous avons amélioré chaque algorithme en proposant des optimisations (utilisation des poids pour tous les NDFS, arrêt plus rapide pour Tau03, intégration des optimisations de SE05 dans Tau03), des heuristiques (pour Cou99), ainsi que des techniques pour le calcul de contre-exemples (pour Cou99 et Tau03).

Dans l'ensemble nos résultats tendent à montrer
- qu'il y a intérêt à travailler avec des automates de Büchi généralisés. Les algorithmes d'*emptiness check* généralisés peuvent vérifier ces automates plus rapidement tout en utilisant moins de mémoire que lorsque l'automate est d'abord dégénéralisé.
- que les algorithmes calculant les CFC sont plus rapides que les NDFS pour tester la vacuité des automates de Büchi généralisés.
- mais que les NDFS (pour automates de Büchi généralisés) gardent un intérêt lorsque l'on souhaite utiliser du *bit-state hashing*.

Il serait à notre avis intéressant d'envisager un algorithme d'*emptiness check* hybride qui combine les deux approches (NDFS et CFC). Un tel algorithme pourrait, dans une première phase, commencer par effectuer un parcours en profondeur pour construire des composantes fortement connexes. Une fois que l'utilisation mémoire aurait dépassé un certain seuil sans que l'algorithme ait conclu, il pourrait entrer dans une seconde phase où il terminerait le parcours de l'automate sur le mode d'un NDFS comme Tau03 opt, plus lent mais moins gourmand. Les CFC partielles calculées pendant la première phase ne seraient pas perdues et pourraient bénéficier à la deuxième.

Chapitre 6

Emptiness check avec inclusion

Ce chapitre est basé sur des travaux que nous présenterons avec Souheib Baarir à la 7e conférence internationale sur l'application de la concurrence à la conception de systèmes (ACSD 2007) [9, 10] ainsi qu'au 6e Colloque Francophone sur la Modélisation des Systèmes Réactifs (MSR'07) [11]. Il introduit deux variantes de l'algorithme d'*emptiness check* du chapitre précédent qui peuvent être utilisées lorsque les états de l'automate à tester représentent des ensembles d'états du système.

6.1 Introduction

L'inconvénient principal de l'approche automate du *model checking* présentée section 2.8 page 28 est l'explosion de l'espace d'état: la taille de l'automate représentant le système croît très rapidement avec la complexité du modèle. Comme cet automate est ensuite synchronisé avec un automate représentant la négation de la propriété à tester, la taille de l'automate produit augmente aussi, ainsi que le temps d'exécution de l'algorithme d'*emptiness check* qui doit tester ce dernier.

Différentes techniques cherchent à réduire la taille de ces automates [134]. Certaines, comme les dépliages [38], les graphes d'observation [68], les réductions basées sur des relations d'équivalences telles que la bisimulation [19], ou encore les réductions basées sur les symétries [67] ont une propriété commune: à la place d'un automate $\mathcal{A}$, elles construisent un automate $\mathcal{B}$ dont les états représentent un ensemble d'états de $\mathcal{A}$ de façon symbolique. Cet automate réduit, testé par un *emptiness check* classique tel que ceux décrits dans le chapitre 5, donnera le même résultat que si l'automate original avait été testé. L'idée est bien sûr de générer $\mathcal{B}$ directement à partir du système et de la propriété à vérifier, sans jamais construire $\mathcal{A}$.

Malheureusement, certaines de ces méthodes peuvent très bien construire un automate « réduit » $\mathcal{B}$ possédant plus d'états que l'automate original $\mathcal{A}$! En effet, si nous appelons $\mathcal{Q}_\mathcal{A}$ et $\mathcal{Q}_\mathcal{B}$ les ensembles d'états de ces deux automates, on a de façon générale $\mathcal{Q}_\mathcal{B} \subseteq 2^{\mathcal{Q}_\mathcal{A}}$ puisque les états de $\mathcal{B}$ représentent des ensembles de ceux de $\mathcal{A}$. Par exemple, la méthode des symétries partielles introduite par Haddad et al. [67] partitionne l'ensemble des successeurs $s_1, \ldots, s_n$ d'un état de l'automate d'origine $\mathcal{A}$ et utilise ces partitions comme les

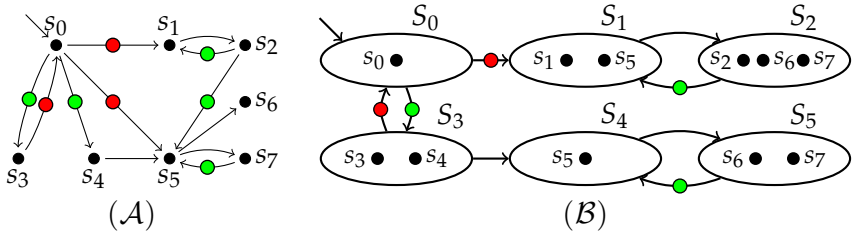


FIG. 6.1: Un automate ensembliste $\mathcal{B}$ pour l'automate $\mathcal{A}$, avec $\mathcal{F}_\mathcal{A} = \mathcal{F}_\mathcal{B} = \{●,●\}$. Les propositions atomiques ne sont pas représentées.
Ces deux automates ne possèdent qu'un chemin acceptant:

$$\mathrm{Acc}(\mathcal{A}) = \{(\langle s_0, \_, ●, s_3\rangle \cdot \langle s_3, \_, ●, s_0\rangle)^\omega\}$$
$$\mathrm{Acc}(\mathcal{B}) = \{(\langle S_0, \_, ●, S_3\rangle \cdot \langle S_3, \_, ●, S_0\rangle)^\omega\}$$

états $S_1, \ldots, S_m$ de l'automate réduit $\mathcal{B}$. Comme cette partition est effectuée localement, pour chaque état, il n'y a aucune garantie qu'un état s de $\mathcal{A}$ soit représenté par un unique état S de $\mathcal{B}$. En pratique, on garde $|\mathcal{Q}_\mathcal{B}| \ll |\mathcal{Q}_\mathcal{A}|$ dans de nombreux cas.

Considérons la figure 6.1. Cette figure montre deux automates $\mathcal{A}$ et $\mathcal{B}$ tels que les états de $\mathcal{B}$ sont des ensembles d'états de $\mathcal{A}$. On remarque qu'un état de $\mathcal{A}$ peut apparaître dans plusieurs états de $\mathcal{B}$ (par exemple l'état s_7 apparaît dans les états S_2 et S_5). Plus important encore, deux états de $\mathcal{B}$ peuvent représenter deux ensembles d'états de $\mathcal{A}$ tels que l'un est inclus dans l'autre (par exemple $S_4 \subseteq S_1$). La possibilité de telles inclusions est à l'origine des algorithmes présentés dans ce chapitre.

Nous montrerons que lorsque l'espace d'état est construit à la volée pendant l'exécution de l'algorithme d'*emptiness check*, il est possible de réaliser des tests d'inclusion pour réduire le nombre d'états construits. Par exemple dans la figure 6.1 nous pouvons éviter de construire les états S_4 et S_5 si nous avons déjà construit S_1 et S_2, puisque les informations qu'ils englobent y sont déjà représentées.

Pour nous abstraire de la technique utilisée pour construire $\mathcal{B}$, nous présenterons nos algorithmes dans le cadre plus général des *automates ensemblistes*, qui sont des TGBA dont les états représentent des ensembles.

La section 6.2 définit ces automates de façon formelle et donne un ensemble de 5 propriétés suffisantes pour lier $\mathcal{A}$ à $\mathcal{B}$ de façon à ce que ces deux automates soient équivalents du point de vue de leur vacuité. La section 6.3 présente notre algorithme d'*emptiness check* pour ces automates ensemblistes. Dans sa forme traditionnelle, cette procédure de décision répond « *vide* » ou « *non vide* ». La section 6.4 montre une petite modification par laquelle l'algorithme devient une procédure de semi-décision qui pourra répondre plus rapidement « *vide* » ou « *je ne sais pas* »,

Les définitions et algorithmes ci-dessus sont génériques au sens où ils ne présument pas de la façon dont $\mathcal{B}$ est construit depuis $\mathcal{A}$. Pour les illustrer par un cas concret, la section 6.5 montre une application à la technique des symétries partielles de Haddad et al. [67]. Nous montrons que la construction de $\mathcal{B}$ avec cette technique vérifie nos hypothèses, puis nous mesurons notre implémentation pour montrer que, bien que théorique-

ment $\mathcal{B}$ puisse avoir $2^{|\mathcal{Q}_\mathcal{A}|}$ états dans le pire des cas, en pratique l'espace d'état visité est effectivement réduit.

6.2 Définitions

Les automates que nous manipulerons sont des TGBA (définition 27 page 48). Comme l'*emptiness check* détermine si $\mathrm{Acc}(\mathcal{A}) = \emptyset$ pour un TGBA $\mathcal{A}$, nous définissons ici une relation d'équivalence entre deux TGBA, basée sur cette propriété.

Définition 37 (Équivalence de vacuité)**.** *Deux TGBA $\mathcal{A}$ et $\mathcal{B}$ sont $\emptyset$-équivalents si et seulement si tous deux ont au moins une exécution acceptante ou aucun n'en a.*

$$\mathcal{A} \overset{\emptyset}{\equiv} \mathcal{B} \quad ssi \quad \mathrm{Acc}(\mathcal{A}) = \emptyset \iff \mathrm{Acc}(\mathcal{B}) = \emptyset$$

Nous proposons maintenant un ensemble de 6 propriétés qui lient deux TGBA $\mathcal{A}$ et $\mathcal{B}$, où les états de $\mathcal{B}$ sont des ensembles d'états de $\mathcal{A}$, de façon à ce que $\mathcal{A}$ et $\mathcal{B}$ soient $\emptyset$-équivalents. À terme, l'objectif est de construire $\mathcal{B}$ en lieu et place de $\mathcal{A}$. Ces propriétés sont vérifiées par les automates de la figure 6.1.

Définition 38 ($\wp$-TGBA)**.** *Soient deux TGBA, $\mathcal{A} = \langle \Sigma_\mathcal{A}, \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}_\mathcal{A}, \delta_\mathcal{A} \rangle$ et $\mathcal{B} = \langle \Sigma_\mathcal{B}, \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}_\mathcal{B}, \delta_\mathcal{B} \rangle$. $\mathcal{B}$ est un* $\wp$-TGBA *sur $\mathcal{A}$ s'il satisfait les propriétés suivantes:*

$$\Sigma_\mathcal{B} = \Sigma_\mathcal{A} \tag{6.1}$$

$$\mathcal{Q}_\mathcal{B} \subseteq 2^{\mathcal{Q}_\mathcal{A}} \setminus \{\emptyset\} \tag{6.2}$$

$$\bigcup_{S \in \mathcal{Q}^0_\mathcal{B}} S = \mathcal{Q}^0_\mathcal{A} \tag{6.3}$$

$$\mathcal{F}_\mathcal{B} = \mathcal{F}_\mathcal{A} \tag{6.4}$$

$$\begin{aligned} &\forall \langle s, P, F, s' \rangle \in \delta_\mathcal{A},\ \forall S \in \mathrm{Reach}(\mathcal{B}), \\ &\qquad s \in S \implies \exists S' \in \mathcal{Q}_\mathcal{B} \text{ tel que } s' \in S' \text{et } \langle S, P, F, S' \rangle \in \delta_\mathcal{B} \end{aligned} \tag{6.5}$$

$$\forall \langle S, P, F, S' \rangle \in \delta_\mathcal{B},\ \forall s' \in S',\ \exists s \in S \text{ tel que } \langle s, P, F, s' \rangle \in \delta_\mathcal{A} \tag{6.6}$$

Proposition 17. *Soient $\mathcal{A} = \langle \Sigma_\mathcal{A}, \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}_\mathcal{A}, \delta_\mathcal{A} \rangle$ et $\mathcal{B} = \langle \Sigma_\mathcal{B}, \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}_\mathcal{B}, \delta_\mathcal{B} \rangle$ deux TGBA tels que $\mathcal{B}$ est un $\wp$-TGBA sur $\mathcal{A}$. Alors $\mathcal{A} \overset{\emptyset}{\equiv} \mathcal{B}$.*

Démonstration. Nous voulons montrer que $\exists \sigma \in \mathrm{Acc}(\mathcal{A}) \iff \exists \sigma' \in \mathrm{Acc}(\mathcal{B})$.

($\Longrightarrow$) Soit $\sigma = \langle s_0, P_0, F_0, s_1 \rangle \cdot \langle s_1, P_1, F_1, s_2 \rangle \cdots \in \mathrm{Acc}(\mathcal{A})$. Puisque $s_0 \in \mathcal{Q}^0_\mathcal{A}$ nous pouvons utiliser (6.3) et trouver un état $S_0 \in \mathcal{Q}^0_\mathcal{B}$ tel que $s_0 \in S_0$. Comme S_0 est accessible dans $\mathcal{B}$ et contient s_0, nous pouvons utiliser (6.5) pour trouver un état $S_1 \in \mathcal{Q}_\mathcal{B}$ tel que $s_1 \in S_1$ et $\langle S_0, P_0, F_0, S_1 \rangle \in \delta_\mathcal{B}$. De la même façon, comme S_1 est accessible dans $\mathcal{B}$ et contient s_1 par construction, nous pouvons utiliser (6.5) à nouveau pour trouver un $S_2 \in \mathcal{Q}_\mathcal{B}$ tel que $s_2 \in S_1$ et $\langle S_1, P_1, F_1, S_2 \rangle \in \delta_\mathcal{B}$. En itérant (6.5) nous pouvons ainsi construire une séquence $\sigma' = \langle S_0, P_0, F_0, S_1 \rangle \cdot \langle S_1, P_1, F_1, S_2 \rangle \cdots \in \mathrm{Run}(\mathcal{B})$ telle que $s_i \in S_i$ pour tout i. De

plus, comme $\mathcal{F}_\mathcal{B} = \mathcal{F}_\mathcal{A}$ (6.4) et que σ' visite chaque condition d'acceptation aussi souvent que σ, $\sigma' \in \mathrm{Acc}(\mathcal{B})$.

($\Longleftarrow$) Soit $\sigma' = \langle S_0, P_0, F_0, S_1\rangle \cdot \langle S_1, P_1, F_1, S_2\rangle \cdots \in \mathrm{Acc}(\mathcal{B})$.

Nous ne pouvons pas utiliser (6.6) directement pour construire un chemin acceptant de $\mathcal{A}$, car il faudrait travailler en remontant depuis la « fin » de la séquence σ' (qui est infinie).

Construisons un arbre dont les nœuds, à l'exception de la racine, sont des états de $\mathcal{A}$. Appelons $\perp$ la racine de l'arbre, à la profondeur 0. Les nœuds de profondeur $n > 0$ sont exactement les états de S_{n-1}. Le père s de tout nœud s' à la profondeur $n > 1$ est choisi parmi les nœuds de profondeur $n-1$ tels que $\langle s, P_{n-1}, F_{n-1}, s'\rangle \in \delta_\mathcal{A}$ (l'équation (6.6) garantit qu'un tel nœud s existe). Le père de tous les nœuds de profondeur 1 est $\perp$. Tous les arcs de cet arbre, à l'exception de ceux quittant la racine, correspondent à des transitions de $\delta_\mathcal{A}$ (grâce à (6.6)).

L'ensemble des nœuds de profondeur $n > 0$ est un sous ensemble de $\mathcal{Q}_\mathcal{A}$, qui est de taille finie. Ainsi, bien que l'arbre construit soit infini, son degré est fini. D'après le lemme de König [86], il contient une branche infinie.

La séquence construite en suivant les arcs de cette branche infinie et en ignorant le premier (quittant $\perp$):

$$\langle s_0, P_0, F_0, s_1\rangle \cdot \langle s_1, P_1, F_1, s_2\rangle \cdots$$

est un chemin acceptant de $\mathcal{A}$. En effet, c'est une exécution de $\mathcal{A}$ ($s_0 \in \mathcal{Q}^0_\mathcal{A}$ d'après (6.3)) qui visite chaque condition d'acceptation aussi souvent que σ'. □

À présent, nous développons deux propositions qui introduisent notre nouvel algorithme d'*emptiness check* sur les $\wp$-TGBA. Rappelons que $\mathcal{A}[\mathcal{T}]$ représente une copie de l'automate $\mathcal{A}$ dont l'ensemble des états initiaux a été remplacé par l'ensemble $\mathcal{T} \subseteq \mathcal{Q}_\mathcal{A}$ (définition 30 page 50).

La première proposition permet d'optimiser l'*emptiness check* si des états sont inclus dans d'autres. Par exemple, dans l'automate $\mathcal{B}$ de la figure 6.1 on a $S_4 \subseteq S_1$: si nous savons qu'il n'existe pas de chemin acceptant qui traverse S_1, il est alors inutile de parcourir les successeurs de S_4 car il ne peut y avoir de chemin acceptant qui traverse cet état.

Proposition 18. *Soit $\mathcal{B} = \langle \Sigma, \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}, \delta_\mathcal{B}\rangle$ un $\wp$-UTGBA sur $\mathcal{A} = \langle \Sigma, \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}, \delta_\mathcal{A}, \rangle$ et soient deux états T et D de $\mathcal{Q}_\mathcal{B}$ tels que $D \subseteq T$. On a*

$$\mathrm{Acc}(\mathcal{B}[\{T\}]) = \emptyset \implies \mathrm{Acc}(\mathcal{B}[\{D\}]) = \emptyset$$

Démonstration. Nous prouvons la contraposée: $\mathrm{Acc}(\mathcal{B}[\{D\}]) \neq \emptyset \implies \mathrm{Acc}(\mathcal{B}[\{T\}]) \neq \emptyset$.

Soit $\sigma = \langle D_0, P_0, F_0, D_1\rangle \cdot \langle D_1, P_1, F_1, D_2\rangle \cdots \in \mathrm{Acc}(\mathcal{B}[\{D\}])$.

En utilisant (6.6) de la même façon que nous l'avons fait dans la partie ($\Longleftarrow$) de la preuve de la proposition 17 page précédente, nous pouvons construire une séquence de transitions de $\mathcal{A}$ $\sigma' = \langle d_0, P_0, F_0, d_1\rangle \cdot \langle d_1, P_1, F_1, d_2\rangle \cdots$ telle que $\forall i \geqslant 0, d_i \in D_i$.

Considérons la première transition de σ': $\langle d_0, P_0, F_0, d_1\rangle$. Puisque $D \subseteq T$, nous avons $d_0 \in T$, par conséquent nous pouvons appliquer la proposition (6.5) pour trouver un

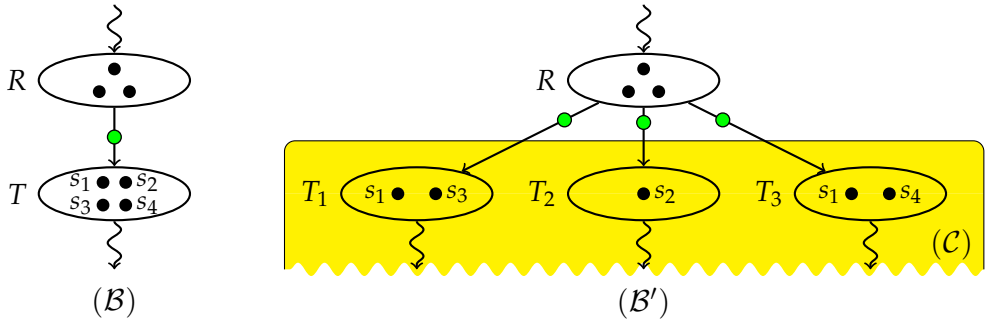


FIG. 6.2: Exemple de décomposition d'une transition $\langle R, \_, \bullet, T\rangle$.

ensemble T_1 tel que $d_1 \in T_1$ et $\langle T, P_0, F_0, T_1\rangle \in \delta_{\mathcal{B}}$. Puis comme $d_1 \in T_1$ par construction nous appliquons la proposition (6.5) à nouveau pour trouver $\langle T_1, P_1, F_1, T_2\rangle \in \delta_{\mathcal{B}}$. En itérant cette opération nous pouvons construire un chemin acceptant: $\sigma'' = \langle T, P_1, F_0, T_1\rangle \cdot \langle T_1, P_1, F_1, T_2\rangle \cdots \in \mathrm{Acc}(\mathcal{B}[\{T\}])$. □

La proposition suivante nous permet de décomposer une transition $\langle R, P, F, T\rangle$ en un ensemble de transitions $\{\langle R, P, F, T_1\rangle, \ldots, \langle R, P, F, T_n\rangle\}$ avec $T_1 \cup \cdots \cup T_n = T$, tout en respectant la $\emptyset$-équivalence. Une telle opération nécessite l'ajout d'états et de transitions à l'automate initial.

Autrement dit, nous voulons substituer à T un automate $\mathcal{C}$ qui a $\{T_1, \ldots, T_n\}$ pour états initiaux et qui est $\emptyset$-equivalent à $\mathcal{A}[T]$. La figure 6.2 illustre cette opération. Cette décomposition est bénéfique si quelques uns des états T_i ont déjà été visités par l'*emptiness check* car il est alors inutile de les visiter à nouveau.

Proposition 19 (Décomposition d'une transition)**.** *Soit $\mathcal{B} = \langle \Sigma, \mathcal{Q}_{\mathcal{B}}, \mathcal{Q}^0_{\mathcal{B}}, \mathcal{F}, \delta_{\mathcal{B}}\rangle$ un $\wp$-TGBA sur $\mathcal{A} = \langle \Sigma, \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}^0_{\mathcal{A}}, \mathcal{F}, \delta_{\mathcal{A}}\rangle$. Soit $\langle R, P, F, T\rangle \in \delta_{\mathcal{B}}$ et $\mathcal{C} = \langle \Sigma, \mathcal{Q}_{\mathcal{C}}, \mathcal{Q}^0_{\mathcal{C}}, \mathcal{F}, \delta_{\mathcal{C}}\rangle$ un $\wp$-TGBA sur $\mathcal{A}[T]$. L'automate $\mathcal{B}' = \langle \Sigma, \mathcal{Q}_{\mathcal{B}} \cup \mathcal{Q}_{\mathcal{C}}, \mathcal{Q}^0_{\mathcal{B}'}, \mathcal{F}, \delta_{\mathcal{B}'}\rangle$ tel que,*

$$\mathcal{Q}^0_{\mathcal{B}'} = \begin{cases} (\mathcal{Q}^0_{\mathcal{B}} \setminus \{T\}) \cup \mathcal{Q}^0_{\mathcal{C}} & \text{si } T \in \mathcal{Q}^0_{\mathcal{B}} \\ \mathcal{Q}^0_{\mathcal{B}} & \text{autrement} \end{cases}$$

$$\delta_{\mathcal{B}'} = (\delta_{\mathcal{B}} \setminus \{\langle R, P, F, T\rangle\}) \cup \{\langle R, P, F, T'\rangle \mid T' \in \mathcal{Q}^0_{\mathcal{C}}\} \cup \delta_{\mathcal{C}}$$

est un $\wp$-TGBA sur $\mathcal{A}$.

Démonstration. Par définition, $\mathcal{B}'$ satisfait les propriétés (6.1), (6.2) et (6.4) de la définition 38 page 145 vis-à-vis de $\mathcal{A}$. Il ne nous reste qu'à prouver que $\mathcal{B}'$ satisfait aussi les propriétés (6.3), (6.5) et (6.6).

(6.3) Nous devons montrer que $\bigcup_{S \in \mathcal{Q}^0_{\mathcal{B}'}} S = \mathcal{Q}^0_{\mathcal{A}}$. Puisque $\mathcal{B}$ est un $\wp$-TGBA sur $\mathcal{A}$, nous avons $\bigcup_{S \in \mathcal{Q}^0_{\mathcal{B}}} S = \mathcal{Q}^0_{\mathcal{A}}$. Si $T \notin \mathcal{Q}^0_{\mathcal{B}}$ nous avons $\mathcal{Q}^0_{\mathcal{B}'} = \mathcal{Q}^0_{\mathcal{B}}$ et la propriété (6.3) est vraie. Sinon, comme $\mathcal{C}$ est un $\wp$-TGBA sur $\mathcal{A}[T]$ nous avons $\bigcup_{S \in \mathcal{Q}^0_{\mathcal{C}}} S = T$, et la propriété (6.3) est aussi vraie.

(6.5) Soit $\langle s, P, F, s'\rangle \in \delta_{\mathcal{A}}$ une transition de $\mathcal{A}$, et $S \in \mathrm{Reach}(\mathcal{B}')$ un état accessible tel que $s \in S$. Pour montrer (6.5) il nous faut trouver une transition $\langle S, P, F, S'\rangle \in \delta_{\mathcal{B}'}$ telle que

$s' \in S'$. Nous distinguons trois cas, qui couvrent toutes les possibilités pour les états de Reach($\mathcal{B}'$):
- Soit $S \in (\text{Reach}(\mathcal{B}) \setminus \{R\}) \vee (S = R \wedge s' \notin T)$. Alors comme $\mathcal{B}$ est un $\wp$-TGBA sur $\mathcal{A}$, la propriété (6.5) assure l'existence d'une telle transition.
- Soit $S = R \wedge s' \in T$. Comme $\mathcal{C}$ est un $\wp$-TGBA sur $\mathcal{A}[T]$, la propriété (6.3) assure que $\exists T' \in \mathcal{Q}^0_{\mathcal{C}}$ tel que $s' \in T'$, par conséquent $\langle S, P, F, T' \rangle \in \delta_{\mathcal{B}'}$ d'après la définition de $\mathcal{B}'$.
- Soit, enfin, $S \in \text{Reach}(\mathcal{C})$. Alors comme $\mathcal{C}$ est un $\wp$-TGBA sur $\mathcal{A}[T]$, la propriété (6.5) appliquée entre $\mathcal{C}$ et $\mathcal{A}$ assure l'existence d'une telle transition.

(6.6) Pour montrer (6.6), pour chaque transition $\langle S, P, F, S' \rangle$ de $\delta_{\mathcal{B}'}$ nous devons nous assurer que $\forall s' \in S', \exists s \in S, \langle s, P, F, s' \rangle \in \delta_{\mathcal{A}}$. Puisque $\mathcal{B}$ est un $\wp$-TGBA sur $\mathcal{A}$ cette propriété est vraie pour toute transition $\langle S, P, F, S' \rangle$ de $\delta_{\mathcal{B}}$. De la même façon, puisque $\mathcal{C}$ est un $\wp$-TGBA sur $\mathcal{A}[T]$, (6.6) est vraie pour toute transition de $\delta_{\mathcal{C}}$. Les transitions restantes, $\{\langle R, P, F, T' \rangle \mid T' \in \mathcal{Q}^0_{\mathcal{C}}\}$ vérifient aussi cette propriété, puisque (6.6) était vraie pour $\langle R, P, F, T \rangle$ et que $T' \subseteq T$. □

6.3 Emptiness check d'un $\wp$-TGBA

Les $\wp$-TGBA étant des TGBA, tous les algorithmes présentés dans le chapitre 5 peuvent être utilisés pour tester si leur langage est vide.

Dans cette section nous proposons un algorithme dédié au $\wp$-TGBA, qui tirera parti des possibilités d'inclusion entre les états. Cet algorithme est une variante de celui basé sur le calcul de CFC que nous avons présenté section 5.4.1 page 120.

Nous ne considérons pas les *emptiness check* basés sur des NDFS (section 5.3 page 108). Les changements que nous proposons guident la construction à la volée de l'automate en effectuant des tests d'inclusion entre chaque nouvel état construit et ceux connus. Comme les NDFS doivent traverser plusieurs fois chaque état, il semble difficile de garantir que le calcul des successeurs sera le même lors d'une visite ultérieure si l'ensemble des états visités a changé. Ce problème se pose aussi pour le calcul d'un contre-exemple, nous y reviendrons section 6.3.3 page 151.

6.3.1 Adaptation aux $\wp$-TGBA

L'algorithme que nous proposons diffère de celui de la figure 5.8 page 121 sur deux points.

Tout d'abord le test des états à retirer est généralisé: tout état D retiré par le DFS de l'algorithme original est aussi retiré par le DFS du nouvel algorithme, mais tout état T vérifiant $T \subseteq D$ peut aussi être retiré. Ceci est possible grâce à la proposition 18.

La figure 6.3 illustre la seconde différence. Considérons l'automate $\mathcal{B}_1$ où le DFS est en train d'examiner la transition $\langle R, P, F, T \rangle$, T étant un nouvel état (non encore visité). Notons qu'il existe un état D dans la pile de recherche (ou plus généralement, dans n'importe quelle CFC de la pile de recherche) tel que $D \subseteq T$. Du point de vue de l'automate sous-jacent $\mathcal{A}$, ceci équivaut à l'existence d'un ensemble d'états dans R qui peuvent at-

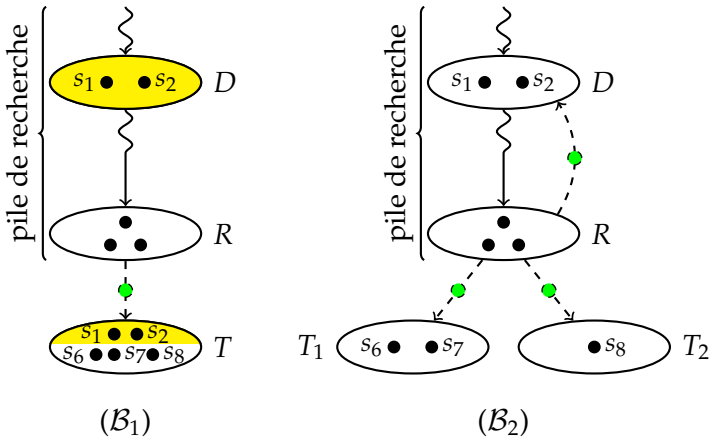


FIG. 6.3: Tests d'inclusion dans la pile. ($D \subseteq T$) On réécrit $\mathcal{B}_1$ en $\mathcal{B}_2$.

teindre des états de D et vice-versa. Par conséquent, ces états de $\mathcal{A}$ appartiennent à la même CFC. Pour l'*emptiness check*, il est bénéfique de diviser la transition $\langle R, P, F, T\rangle$ pour obtenir un automate ressemblent à $\mathcal{B}_2$ (les successeurs $T_1, T_2, \ldots$ peuvent être plus ou moins nombreux): cet automate rend explicite la boucle sur la CFC et réutilise les états déjà visités. Une telle décomposition est correcte du fait de la proposition 19, mais nécessite des contraintes supplémentaires, que nous formalisons ci-dessous.

Soit $\mathcal{B} = \langle \Sigma, \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}, \delta_\mathcal{B}\rangle$ un ℘-TGBA sur $\mathcal{A}$. L'opération Decomp$(\mathcal{B}, \langle S, P, F, T\rangle, D)$ x réalise une décomposition respectant la proposition 19. En plus de $\mathcal{B}$ et $\langle S, P, F, T\rangle$, qui ont la même signification que dans la proposition, D est un état de $\mathcal{B}$ tel que $D \subseteq T$. Decomp construit $\mathcal{C}$ avec deux contraintes supplémentaires:
(1) on veut $D \in \mathcal{Q}^0_\mathcal{C}$ (les autres états de $\mathcal{Q}^0_\mathcal{C}$ vont, par définition, compléter T) ;
(2) $\delta_\mathcal{C}$ ne doit pas ajouter de transitions pour les états de $\mathcal{B}$, *i.e.* $\{\langle S, P, F, S'\rangle \in \delta_\mathcal{C} \mid S \in \text{Reach}(\mathcal{B})\} \subseteq \delta_\mathcal{B}$. Decomp retourne une paire $\mathcal{B}', \mathcal{Q}^0_\mathcal{C}$: le nouvel automate, et les états initiaux de $\mathcal{C}$.
Puisqu'en pratique nous construisons $\mathcal{B}$ à la volée et que Decomp ne rajoute pas de transitions à la partie de $\mathcal{B}$ déjà visitée par l'*emptiness check*, l'algorithme peut considérer l'automate issu de Decomp comme s'il s'agissait de l'automate de départ. Ceci justifie la seconde contrainte. Quant à la première, il s'agit simplement de réaliser notre objectif: dans la figure 6.3 nous voulons décomposer la transition $\langle R, \_, \bullet, T\rangle$ de l'automate $\mathcal{B}_1$ pour faire apparaître la transition $\langle R, \_, \bullet, D\rangle$ de l'automate $\mathcal{B}_2$. Il faut donc que D soit l'un des nouveaux états successeurs..

6.3.2 Nouvel algorithme

La figure 6.4 montre le listing complet de l'*emptiness check* d'un ℘-TGBA. Seules les lignes 11–22 diffèrent de l'original.

Le schéma de preuve est le même que celui que nous avons suivi section 5.4.1 page 120: il faut prouver les invariants 12–16 de la page 124 sur chaque ligne de la fonction "Main".

```
1  Entrée : Un TGBA B = ⟨Σ_B, Q_B, q⁰_B, F_B, δ_B⟩
2  Résultat : ⊤ si et seulement si L(B) = ∅
3  Données :  todo: stack of ⟨state ∈ Q_B, succ ⊆ δ_B⟩
              SCC: stack of ⟨root ∈ ℕ, la ⊆ F_B, acc ⊆ F_B, rem ⊆ Q_B⟩
              H: map of Q_B ↦ ℕ
              max ← 0
4  begin
5      forall S⁰ ∈ Q⁰_B do
6          DFSpush(∅, S⁰_B)
7          while ¬todo.empty() do
8              if todo.top().succ = ∅ then
9                  DFSpop()
10             else
11                 pick one ⟨T, P, F, D⟩ off todo.top().succ
12                 if ∃D ∈ H.keys() such that (T ⊆ D) ∧ H[D] = 0 then
13                     continue
14                 else if T ∉ H then
15                     if ∃D ∈ H.keys() such that (D ⊆ T) ∧ H[D] > 0 then
16                         B, Q⁰_C ← Decomp(B, ⟨R, P, F, T⟩, D)
17                         todo.top().succ ← todo.top().succ ∪ {⟨R, P, F, D⟩} ∪ {(R, P, F, T′) |
                           T′ ∈ Q⁰_C}
18                     else
19                         DFSpush(F,T)
20                 else if H[T] > 0 then
21                     merge(F, H[T])
22                     if SCC.top().acc = F then return ⊥
23     return ⊤
24 end
25 DFSpush(F ⊆ F, S ∈ Q)
26     max ← max + 1
27     H[S] ← max
28     SCC.push(⟨max, F, ∅, ∅⟩)
29     todo.push(⟨S, {⟨R, L, F, T⟩ ∈ δ | R = S}⟩)
30 end
31 DFSpop()
32     ⟨S, _⟩ ← todo.pop()
33     SCC.top().rem.insert(S)
34     if H[S] = SCC.top().root then
35         foreach R ∈ SCC.top().rem do
36             H[R] ← 0
37         SCC.pop()
38 end
39 merge(F ⊆ F, n ∈ ℕ)
40     r ← ∅
41     while n < SCC.top().root do
42         F ← a ∪ SCC.top().acc ∪ SCC.top().la
43         r ← r ∪ SCC.top().rem
44         SCC.pop()
45     SCC.top().acc ← SCC.top().acc ∪ F
46     SCC.top().rem ← SCC.top().rem ∪ r
47 end
```

FIG. 6.4: *Emptiness check* d'un $\wp$-TGBA.

Une preuve complète peut être trouvée dans notre rapport technique [9], nous ne la reproduisons pas à cause de sa similarité avec celle de la section 5.4.1.

Les seules différences sont celles qui découlent des tests d'inclusion:
– Ligne 12, les états qui sont inclus dans des états retirés peuvent être ignorés grâce à la proposition 18 page 146.
– Lorsqu'une transition arrive sur un nouvel état qui inclut un état actif, ligne 15, nous décomposons cette transition avec la propriété 19 pour faire apparaître l'état en question. Comme nous avons supposé que cette décomposition n'ajoutait pas de transition à des états visités, nous pouvons substituer l'automate sur lequel l'algorithme travaille par le nouvel automate (où la transition est décomposée) et continuer l'algorithme comme s'il avait travaillé avec cet automate depuis le début. (En pratique, comme nous construisons l'automate à la volée, il ne s'agit pas réellement d'une substitution d'automate.)

Ce nouvel algorithme possède quelques points critiques qui devront être résolus par une bonne implémentation ainsi qu'un choix judicieux de la représentation des « états ensemblistes ».
– De façon générale, nous avons besoin de pouvoir tester l'inclusion entre deux états assez rapidement.
– Ligne 12 et 15, nous cherchons parmi les états visités des états qui incluent ou sont inclus dans un état donné. Une recherche dans tous les états visités serait très coûteuse, il semble indispensable de disposer d'un mécanisme pour réduire le champ de cette recherche. Notons qu'il n'est pas nécessaire que la recherche soit complète: l'algorithme reste correct si une inclusion possible est ignorée.
– La décomposition d'une transition doit être possible sous les contraintes listées section précédente.

Les choix que nous avons faits pour notre implémentation de test seront discutés section 6.5.2 page 155.

6.3.3 Contre-exemples

Comme nous l'avons vu dans le chapitre précédent, les algorithmes d'*emptiness check* basés sur les CFC ne fournissent pas directement un contre-exemple (chemin acceptant) quand ils détectent un automate non vide.

La méthode que nous avons proposée section 5.6.1 page 137 ne fonctionne que si l'on peut garantir que plusieurs parcours de l'automate traversent les mêmes états. Dans le cas présent cela serait faussé par les utilisations de la décomposition et de l'inclusion. En effet, les décompositions que nous faisons lors de l'exploration de l'automate dépendent des états apparaissant dans H: si pour calculer un contre-exemple nous parcourons l'automate à nouveau avec un tableau associatif H différent, les inclusions effectuées ne seront pas forcément les mêmes et cela pourrait créer de nouveaux états.

D'autre part, il est sans doute plus intéressant pour l'utilisateur d'obtenir un contre-exemple dans l'automate $\mathcal{A}$ plutôt que dans $\mathcal{B}$. Un contre-exemple sur $\mathcal{A}$ sera plus proche du système vérifié, alors que les états de $\mathcal{B}$ sont des ensembles (représentés de façon symbolique) dont l'interprétation n'est pas forcément très aisée.

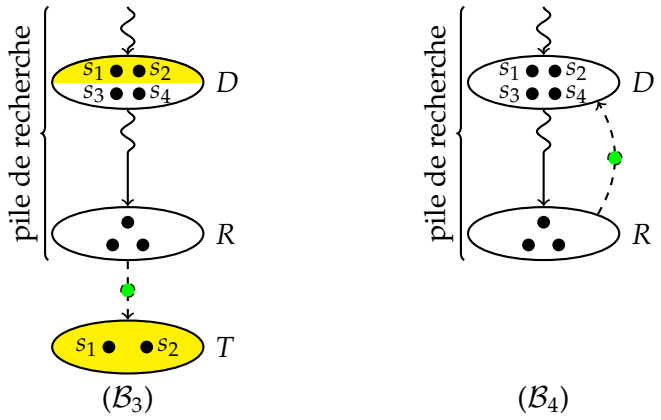


FIG. 6.5: Tests d'inclusion dans la pile de recherche, où $\mathcal{B}_3$ est réécrit en $\mathcal{B}_4$.

Nous suggérons donc de calculer un contre-exemple sur $\mathcal{A}$ en utilisant les structures de données générées par l'algorithme d'*emptiness check* sur $\mathcal{B}$ pour restreindre l'espace de recherche. En parcourant les piles *todo* et *SCC* nous pouvons reconstruire les états appartenant à chaque CFC. L'ensemble des états de $\mathcal{B}$ qui appartiennent à la dernière CFC est $\mathfrak{S}_n = rem_n \cup \{state_{f(root_n)}, state_{f(root_n)+1}, \ldots, state_m\}$. Les états de la CFC précédente sont alors $\mathfrak{S}_{n-1} = rem_{n-1} \cup \{state_{f(root_{n-1})}, state_{f(root_{n-1})+1}, \ldots, state_{f(root_n)-1}\}$, etc. Pour savoir si un état $s \in \mathcal{Q}_\mathcal{A}$ appartient à la i^{e} CFC il suffit de chercher si $\exists \mathfrak{S} \in \mathfrak{S}_i$ tel que $s \in \mathfrak{S}$. Nous pouvons donc appliquer la technique de la section 5.6.1 pour construire un contre exemple dans $\mathcal{A}$ en utilisant les données construites par l'*emptiness check* de $\mathcal{B}$.

6.4 *Emptiness check* semi-décisionnel

Considérons à nouveau la figure 6.3. Nous avons vu que les lignes 18–20 de l'algorithme transforment la situation illustrée par l'automate $\mathcal{B}_1$, où l'algorithme atteint l'état $T \supseteq D$ tel que D appartient à la pile de recherche, en celle illustrée par l'automate $\mathcal{B}_2$. Cette transformation cherche à réutiliser les états existants et par conséquent à construire des CFC le plus rapidement possible. Nous avons prouvé que cette transformation préserve le résultat de l'*emptiness check* ($\mathcal{B}_1 \overset{\emptyset}{\equiv} \mathcal{B}_2$).

À présent, nous voulons examiner la situation illustrée par l'automate $\mathcal{B}_3$ de la figure 6.5, où l'*emptiness check* traite une transition $\langle R, P, F, T\rangle$ telle que $T \subseteq D$ et D est dans la pile de recherche. Nous voulons remplacer cette transition par $\langle R, P, F, D\rangle$ comme illustré dans $\mathcal{B}_4$. Sur l'algorithme précédent, il suffit de substituer les lignes 15–17 par celles données figure 6.6 page ci-contre.

Considérons que $\mathcal{B}_3$ est un $\wp$-TGBA sur $\mathcal{A}$. On note que la transformation ci-dessus ne préserve pas la propriété (6.6) de la définition 38, car s_3 et s_4 n'ont pas de prédécesseur dans R ; donc $\mathcal{B}_4$ *n'est pas un* $\wp$-TGBA sur $\mathcal{A}$. Cependant, en ajoutant des transitions à $\mathcal{A}$ pour respecter la propriété (6.6), il est possible de dériver un automate $\mathcal{A}'$ tel que $\mathcal{B}_4$ soit un $\wp$-TGBA sur $\mathcal{A}'$. Par conséquent, si l'algorithme d'*emptiness check* trouve une CFC

```
15 if ∃D ∈ H.keys() such that (T ⊆ D) ∧ H[D] > 0 then
16 |  /* Note the order of T and D above.                    */
17 |  todo.top().succ ← todo.top().succ ∪ {⟨R, P, F, D⟩} ∪ {(R, P, F, T′) | T′ ∈ Q_C^0}
18
```

FIG. 6.6: *Emptiness check* approximatif: lignes à remplacer dans la figure 6.4 page 150.

acceptante dans $\mathcal{B}_4$, alors il existe un chemin acceptant dans $\mathcal{A}'$ *mais pas nécessairement* dans $\mathcal{A}$. Cependant, puisque les chemins de $\mathcal{A}$ sont aussi des chemins de $\mathcal{A}'$ alors si l'algorithme ne trouve pas de chemin acceptant dans $\mathcal{B}_4$, *forcément*, il n'y aura de chemin acceptant ni dans $\mathcal{A}$, ni dans $\mathcal{A}'$.

Autrement dit, cet algorithme modifié retourne « *vide* » ou « *Je ne sais pas* ». Cette manière, *semi-décisionnelle*, de procéder s'avère très utile dans les cas où nous sommes *presque sûrs* de la véracité d'une propriété et que nous souhaitons une preuve rapide.

6.5 Application aux symétries

Dans cette section, nous montrons comment exploiter les symétries du modèle pour construire un TGBA $\mathcal{B}$ qui est un $\wp$-TGBA sur $\mathcal{A}$. En pratique nos modèles sont modélisés par des réseaux de Petri bien formés, une classe de réseaux de Petri colorés qui permet de calculer facilement ces symétries [28, 27, 67, 7]. Pour éviter d'introduire un nouveau formalisme, nous nous contenterons de présenter cette méthode sur un *système de transitions étiquetées*. (Le lecteur familier avec les réseaux de Petri peut voir ce système de transitions comme notre interprétation du graphe des marquages accessibles.)

Nous allons donc définir $\mathcal{A}$, le produit synchronisé d'un *système de transitions étiquetés* $\mathcal{T}$, représentant le comportement du système, avec un TGBA $\mathcal{P}$, représentant la propriété à tester.

Définition 39 (Système de transitions étiquetées)**.** *Un* système de transitions étiquetées *est un quadruplet* $\mathcal{T} = \langle \Sigma, \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \delta_\mathcal{T} \rangle$*, où:*
– $\Sigma = 2^{AP}$ *est un alphabet, où* AP *est l'ensemble des propositions atomiques ;*
– $\mathcal{Q}_\mathcal{T}$ *est un ensemble fini d'états ;*
– $\mathcal{Q}^0_\mathcal{T} \subseteq \mathcal{Q}_\mathcal{T}$ *est l'ensemble des états initiaux ;*
– $\delta_\mathcal{T} \subseteq \mathcal{Q}_\mathcal{T} \times \Sigma \times \mathcal{Q}_\mathcal{T}$ *est une relation de transition telle que* $\forall \langle s_1, p_1, d_1 \rangle \in \delta_\mathcal{T}, \forall \langle s_2, p_2, d_2 \rangle \in \delta_\mathcal{T},\ p_1 = p_2 \iff (s_1, d_1) = (s_2, d_2)$.
Les états accessibles $\mathrm{Reach}(\mathcal{T})$ *sont définis classiquement.*

Si l'on omet la dernière condition de $\delta_\mathcal{T}$, on peut facilement voir ce système comme un TGBA sans condition d'acceptation et avec une seule étiquette par transition.

La condition sur $\delta_\mathcal{T}$ implique que chaque transition est définie de façon unique par son étiquette (notons que nous pouvons toujours ajouter des propositions atomiques pour satisfaire cette contrainte). Cette unicité sera indispensable pour définir la notion de transition symétrique (définition 42 page suivante).

Définition 40 (Produit synchronisé). *Le produit synchronisé entre le système de transitions étiquetées $\mathcal{T} = \langle \Sigma, \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \delta_\mathcal{T} \rangle$ et le TGBA $\mathcal{P} = \langle \Sigma, \mathcal{Q}_\mathcal{P}, \mathcal{Q}^0_\mathcal{P}, \mathcal{F}, \delta_\mathcal{P} \rangle$ est le TGBA $\mathcal{A} = \mathcal{T} \otimes \mathcal{P}$ défini par $\mathcal{A} = \langle \Sigma, \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}, \delta_\mathcal{A} \rangle$, où:*
- $\mathcal{Q}_\mathcal{A} = \mathcal{Q}_\mathcal{T} \times \mathcal{Q}_\mathcal{P}$;
- $\mathcal{Q}^0_\mathcal{A} = \mathcal{Q}^0_\mathcal{T} \times \mathcal{Q}^0_\mathcal{P}$;
- $\delta_\mathcal{A} \subseteq \mathcal{Q}_\mathcal{A} \times 2^\Sigma \times 2^\mathcal{F} \times \mathcal{Q}_\mathcal{A}$ *est la relation de transition telle que* $\exists \langle \langle s, q \rangle, \{p\}, F, \langle s', q' \rangle \rangle \in \delta_\mathcal{A}$ *si et seulement si* $\exists \langle s, p, s' \rangle \in \delta_\mathcal{T}, \exists \langle q, P', F, q' \rangle \in \delta_\mathcal{P}$ *et* $p \in P'$.

6.5.1 Construction d'un $\wp$-TGBA basée sur les symétries

Nous construisons maintenant $\mathcal{B}$, un $\wp$-TGBA sur $\mathcal{A} = \mathcal{T} \otimes \mathcal{P}$, en utilisant la technique introduite par Haddad et al. [67] mais adaptée aux automates basés sur les transitions. L'idée est d'exploiter les symétries de $\mathcal{T}$ en plus de celles des arcs de $\mathcal{P}$, pour regrouper les nœuds de $\mathcal{A}$.

Puisque les symétries s'appuient sur la théorie des groupes, nous en rappelons quelques notions utiles, afin de caractériser un système de transition symétrique.

Définition 41. *Soient $(\mathcal{G}, \circ)$ un groupe avec un élément neutre id et E un ensemble.*
- *Une action de $\mathcal{G}$ sur E est une application $\mathcal{G} \times E \mapsto E$ telle que l'image de $\langle g, e \rangle$, notée $g.e$, vérifie $\forall e \in E : id.e = e$ et $\forall g, g' \in \mathcal{G}, (g \circ g').e = g.(g'.e)$.*
- *Le sous-groupe d'isotropie $\mathcal{G}_{E'}$ pour un sous-ensemble $E' \subseteq E$ est défini par $\mathcal{G}_{E'} = \{g \in \mathcal{G} \mid \forall e \in E', g.e \in E'\}$.*
- *Pour un sous-groupe $\mathcal{H}$ de $\mathcal{G}$ (noté $\mathcal{H} < \mathcal{G}$), l'orbite $\mathcal{H}.e$ de $e \in E$ sous $\mathcal{H}$ est $\mathcal{H}.e = \{g.e \mid g \in \mathcal{H}\}$.*
- *Une action g de $\mathcal{G}$ peut être étendue facilement à l'ensemble des parties de E. Pour tout $E' \subseteq E$, $g.E' = \{g.e \mid e \in E'\}$.*

Définition 42 (Système de transition symétrique). *Soit $\mathcal{T} = \langle \Sigma, \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \delta_\mathcal{T} \rangle$ un système de transition et $\mathcal{G}$ un groupe agissant sur AP. $\mathcal{T}$ est dit* symétrique par rapport à *$\mathcal{G}$ si et seulement si chaque transition de $\mathcal{T}$ a une transition "symétrique" par rapport à n'importe quel élément de $\mathcal{G}$ et l'action de $\mathcal{G}$ est congruente par rapport à la relation de transition: $\forall g \in \mathcal{G}$, $\forall \langle s_1, p_1, d_1 \rangle \in \delta_\mathcal{T}, \exists \langle s_2, p_2, d_2 \rangle \in \delta_\mathcal{T}$ telle que*

$$\begin{cases} s_1 \in \mathcal{Q}^0_\mathcal{T} \iff s_2 \in \mathcal{Q}^0_\mathcal{T}, \\ p_2 = g.p_1 \text{ et} \\ \forall \langle s'_1, p'_1, d'_1 \rangle \in \delta_\mathcal{T}, s'_1 = d_1, \exists \langle s'_2, p'_2, d'_2 \rangle \in \delta_\mathcal{T}, \text{ t.q. } s'_2 = d_2 \text{ et } p'_2 = g.p_2 \end{cases}$$

L'action du groupe sur AP est étendue à Reach$(\mathcal{T})$ *en notant $g.s$ l'unique s_2 tel que $\forall g \in \mathcal{G}, \forall \langle s_1, p_1, d_1 \rangle \in \delta_\mathcal{T}, s_1 = s, \exists d_2 \in \mathcal{Q}_\mathcal{T}, \langle s_2, g.p_1, d_2 \rangle \in \delta_\mathcal{T}$. (L'unicité est assurée par la contrainte sur $\delta_\mathcal{T}$ dans la définition 39 page précédente.)*
Comme $\mathcal{G}$ est un groupe, une conséquence de cette définition est que $\mathcal{G}.\mathcal{Q}^0_\mathcal{T} = \mathcal{Q}^0_\mathcal{T}$.

Ces définitions nous permettent de proposer une construction de $\mathcal{B}$.

Définition 43 (Produit Synchronisé Symbolique). *Soit $\mathcal{T} = \langle \Sigma, \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \delta_\mathcal{T} \rangle$ un système de transitions symétrique par rapport à un groupe $\mathcal{G}$ et $\mathcal{P} = \langle \Sigma, \mathcal{Q}_\mathcal{P}, \mathcal{Q}^0_\mathcal{P}, \mathcal{F}, \delta_\mathcal{P} \rangle$ un TGBA. Le* Produit Synchronisé Symbolique *de $\mathcal{T}$ et $\mathcal{P}$ est un TGBA $\mathcal{B} = \langle \Sigma, \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}, \delta_\mathcal{B} \rangle$ où:*

- $\mathcal{Q}_\mathcal{B}$ *est l'ensemble fini des n-uplets de la forme* $\langle \mathcal{H}, O, q \rangle$ *tels que* $\mathcal{H} < \mathcal{G}$, $O \subseteq \text{Reach}(\mathcal{T})$, $q \in \mathcal{Q}_\mathcal{P}$ *et* $\mathcal{H}.O = O$.
- $\mathcal{Q}^0_\mathcal{B} = \{\langle \mathcal{G}, \mathcal{G}.s, q \rangle \mid s \in \mathcal{Q}^0_\mathcal{T}, q \in \mathcal{Q}^0_\mathcal{P}\}$
- $\delta_\mathcal{B}$ *est défini par construction comme suit:* $\big\langle \langle \mathcal{H}, O, q \rangle, \{p,\}, F, \langle \mathcal{H}', O', q' \rangle \big\rangle \in \delta_\mathcal{B}$ *si et seulement si* $\exists (s, s', P', F) \in O \times O' \times 2^\Sigma \times 2^\mathcal{F}$ *tel que* $\langle s, p, s' \rangle \in \delta_\mathcal{T}$, $\langle q, P', F, q' \rangle \in \delta_\mathcal{P}$ *et* $p \in P'$. *Alors* $O' = (\mathcal{H} \cap \mathcal{G}_p).s'$ *et* $\mathcal{H}' \subseteq \mathcal{G}_{O'}$.

Si $\mathcal{A} = \mathcal{T} \otimes \mathcal{P} = \langle \Sigma, \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}, \delta_\mathcal{A} \rangle$, tout état $\langle \mathcal{H}, O, q \rangle \in \mathcal{Q}_\mathcal{B}$ de $\mathcal{B}$ représente les états $\{\langle x, q' \rangle \in \mathcal{Q}_\mathcal{A} \mid x \in O \wedge q' = q\}$ de $\mathcal{A}$. Par conséquent nous pouvons écrire $\langle x, q' \rangle \in \langle \mathcal{H}, O, q \rangle$, et prouver que $\mathcal{B}$ est un $\wp$-TGBA sur $\mathcal{A}$ [9].

La méthode, telle que nous l'avons présentée jusqu'ici, est dépendante des symétries de $\mathcal{T}$ (*i.e.* du groupe $\mathcal{G}$). Plus $\mathcal{G}$ est gros, meilleure est la réduction. Sur un système de transition globalement asymétrique, $\mathcal{G}$ sera très petit (si $\mathcal{G} = \{id\}$ il n'y a aucune réduction) et les sous-groupes $\mathcal{H} < \mathcal{G}$ calculés pour chaque nœud autoriseront moins de réduction.

Une manière de contourner ce problème sera de réécrire le système $\mathcal{T}$ sous la forme d'une composition $\mathcal{T} = \mathcal{T}_S \otimes \mathcal{C}$ entre un automate $\mathcal{T}_S$ symétrique par rapport à un large $\mathcal{G}$ et un automate $\mathcal{C}$ dit *de contrôle*. Au lieu de construire le produit synchronisé symbolique représentant $\mathcal{T} \otimes \mathcal{P}$, nous construisons celui représentant $\mathcal{T}_S \otimes \mathcal{P}_C$ où $\mathcal{P}_C = \mathcal{C} \otimes \mathcal{P}$. Nous transférons donc les asymétries du système vers $\mathcal{P}_C$. Cela fonctionne parce que la méthode ne requiert pas de symétries globales sur l'automate de la propriété.

Haddad et al. [67] montrent une façon de construire $\mathcal{T}_S$ et $\mathcal{C}$ à partir de $\mathcal{T}$, alors que nous utilisons une méthode plus élaborée développée par Baarir [8], Chapitres 4 et 7.

6.5.2 Structures et opérations pour l'*emptiness check*

Du point de vue de l'occupation mémoire, cette construction n'a de sens que si l'ensemble O d'un état $\langle \mathcal{H}, O, q \rangle$ n'est pas représenté explicitement. Dans notre implémentation, nous utilisons pour cela une version modifiée de la représentation symbolique utilisée par les réseaux de Petri bien formés [7]. Cette structure de données supporte les tests d'inclusion.

La recherche, parmi les états visités, d'états qui en incluent d'autres (lignes 15 et 18 de la figure 6.4) est une opération coûteuse: naïvement, il faudrait effectuer un test d'inclusion avec tous les états déjà visités par l'algorithme. Nous l'avons accélérée en utilisant une table de hachage à deux niveaux.

Soit $\mathcal{G}$ le groupe agissant sur AP tel que $\forall p \in AP, \mathcal{G}.p = AP$. Pour un état $\langle \mathcal{H}_1, O_1, q_1 \rangle$, choisissons $s \in O_1$; $\mathcal{G}.s$ est la plus grande classe d'équivalence à laquelle s peut appartenir. Tous les états susceptibles d'inclure ou d'être inclus par $\langle \mathcal{H}_1, O_1, q_1 \rangle$ ont nécessairement les mêmes $\mathcal{G}.s$ et q_1. En utilisant $\mathcal{G}.s$ et q_1 comme clefs de hachage, nous pouvons partitionner les états au fur et à mesure qu'ils sont découverts, de façon à faciliter la recherche des inclusions par la suite.

6.5.3 Évaluation

Le produit synchronisé symbolique de la définition 43 a été implémenté en réutilisant le noyau de GreatSPN (`http://www.di.unito.it/~greatspn/`) pour développer l'es-

				SP+Cou99			SSP+Cou99		
modèle		n		ét.	tr.	T	ét.	tr.	T
WCS3	asym.	22	produit vide	99	279	0.06	94	255	0.13
WCS4	asym.	22	produit vide	434	1485	0.13	602	2063	1.60
WCS5	asym.	22	produit vide	1889	7430	0.60	4224	17744	144
PO22	sym.	32	produit vide	2484	5482	0.91	866	1817	2.16
PO23	sym.	32	produit vide	3253	7200	1.56	952	2030	3.68
PO32	sym.	28	produit vide	4617	10651	2.48	1334	2848	4.97
WCS3	asym.	28	prod. non vide	28	80	0.05	25	58	0.06
WCS4	asym.	28	prod. non vide	78	250	0.06	77	202	0.14
WCS5	asym.	28	prod. non vide	290	979	0.13	416	1309	2.76
PO22	sym.	18	prod. non vide	252	431	0.13	690	1307	0.95
PO23	sym.	18	prod. non vide	292	511	0.17	770	1441	1.48
PO32	sym.	22	prod. non vide	1173	2235	0.65	2184	4730	7.40

	SSP+NSIEC			SSP+IEC			SSP+SDEC		
modèle	ét.	tr.	T	ét.	tr.	T	ét.	tr.	T
WCS3	91	250	0.14	73	194	0.13	30	70	0.07
WCS4	568	1980	2.17	297	940	1.07	64	177	0.15
WCS5	3905	16719	107	1370	4815	23.7	136	428	0.46
PO22	864	1814	2.14	865	1814	2.14	864	1813	2.14
PO23	868	1830	3.08	868	1831	3.09	868	1830	3.08
PO32	1294	2784	4.78	1294	2784	4.78	1294	2783	4.79
WCS3	26	51	0.06	24	45	0.05	21	39	0.05
WCS4	66	176	0.17	43	96	0.10	29	57	0.06
WCS5	294	1118	6.44	106	287	0.95	39	82	0.07
PO22	738	1505	1.10	738	1505	1.10	738	1505	1.10
PO23	750	1550	1.69	750	1550	1.69	750	1550	1.69
PO32	1400	3031	3.75	1400	3031	3.75	1392	2982	3.71

TAB. 6.1: États (ét.) et transitions (tr.) explorés par chaque algorithme sur différents modèles et temps (T) passé. Moyennes sur n propriétés.

pace d'état. Les algorithmes d'*emptiness check* que nous avons présentés sont implémentés dans Spot (annexe A page 201).

Le tableau 6.1 présente des résultats obtenus sur deux modèles paramétrés: WCS [7] et PO [79]. PO est un modèle complètement symétrique (les objets de nature similaire se comportent identiquement) alors que WCS est asymétrique (les objets se comportent différemment). Dans les deux cas, augmenter les paramètres élargit l'espace d'état. Chacun de ces modèles a été synchronisé avec 50 automates représentant des propriétés. Ces propriétés sont plus ou moins aléatoires: nous avons choisi les propositions atomiques qui pouvaient être observées sur les deux modèles puis généré de façon aléatoire 50 formules LTL les utilisant. Les formules ont une taille comprise entre 7 et 10 après simplification[1] et sont uniques. Les mêmes formules sont utilisées pour les trois variantes de chaque modèle.

Nous avons séparé les résultats où le produit est vide (l'*emptiness check* ne trouve pas d'exécution acceptante) de ceux où il contient une exécution acceptante. Cette distinction

[1] La simplification est faite par l'ensemble des techniques présentées dans la section 4.3.3 page 94

est nécessaire pour l'interprétation des résultats, car dans le cas d'un produit vide l'algorithme doit parcourir la totalité des états du produit, alors que pour un produit non vide il se termine dès la découverte de la première SCC acceptante. La colonne *n* indique combien de ces 50 produits étaient vides ou non.

Les abréviations dans les en-têtes des cinq colonnes indiquent la manière dont le produit a été construit, ainsi que l'*emptiness check* utilisé. SP est le produit synchronisé de la définition 40 page 154 tandis que SSP est le produit synchronisé de la définition 43 page 154. Les états de SP ne sont pas des ensembles, donc cet automate est vérifié par l'algorithme d'*emptiness check* de la section 5.4.1 page 120 (Cou99). Rappelons que Cou99 est similaire à l'algorithme présenté dans ce chapitre (figure 6.4 page 150) mais sans les tests d'inclusion et les décompositions. IEC désigne l'*emptiness check* de la figure 6.4. NSIEC est le même algorithme sans les lignes 15–17 (c'est-à-dire sans test d'inclusion dans la pile de recherche). Finalement, SDEC désigne l'*emptiness check* semi-décisionnel de la section 6.4.

Comparons tout d'abord les deux premières colonnes, où les *emptiness checks* présentés dans ce chapitre ne sont pas employés. Elles montrent assez bien que le produit synchronisé symbolique (SSP) peut donner des résultats pires que le produit synchronisé classique (SP), même s'il est présenté comme une technique de réduction de l'espace d'état. Ici l'espace d'état est plus gros dans la moitié des huit cas présentés. Cependant quatre de ces cas correspondent à des produits non vides où l'algorithme d'*emptiness check* peut s'arrêter plus où moins rapidement selon la chance avec laquelle il trouve un contre-exemple. Il est donc difficile de conclure de ce côté. Les lignes correspondant aux produits vides des modèles WCS4 et WCS5 montrent clairement des cas où le processus de vérification a passé plus de temps à construire puis explorer un produit plus gros.

La comparaison entre SSP+Cou99 et la colonne suivante (SSP+NSIEC) montre l'apport de la proposition 18 page 146. En ignorant, lors de l'*emptiness check*, les états qui sont inclus dans des états à partir desquels nous savons qu'il n'existe aucun chemin acceptant, nous réduisons la taille de l'espace d'état construit par SSP systématiquement, mais pas encore au point de gagner sur SP. Les deux cas où NSIEC augmente le nombre d'états visités correspondent à des produits non vides ; ils s'expliquent par le fait que l'ordre de parcours de l'automate est modifié à partir du moment où un état est ignoré et que ses successeurs ne sont pas construits: la découverte d'un chemin acceptant peut donc être plus longue.

La colonne SSP+IEC ajoute la décomposition des transitions dont la destination inclut un état de la pile de recherche (figure 6.3 page 149). Avec ce nouvel algorithme, les cas correspondant à des produits possèdent maintenant tous un nombre d'états moindre que le produit construit par SP+Cou99. Même si l'opération de décomposition est coûteuse (en temps), elle est indispensable à la réduction du nombre d'états (donc de la consommation mémoire) dont le parcours plus rapide compense la perte de temps. Sur le modèle PO ce nouvel algorithme n'a que très peu d'effet car le modèle est symétrique et les tests d'inclusion ne se produisent que très rarement ; on constate cependant que ces tests d'inclusion additionnels, même s'ils échouent, n'induisent aucune perte de temps significative.

La dernière colonne (SSP+SDEC) donne les résultats pour l'algorithme semi-décisionnel présentés dans la section 6.4 page 152. La vérification du modèle WCS est améliorée dans tous les cas ; tandis que le modèle PO reste stable pour la raison donnée dans le paragraphe précédent.

6.6 Conclusion

Nous avons présenté deux nouveaux algorithmes d'*emptiness check* pour des automates dont les états représentent des ensembles, en exploitant les inclusions qui peuvent exister entre ces ensembles.

Nos résultats sur la construction basée sur les symétries indiquent que ces tests d'inclusion, ainsi que la décomposition, permettent de réduire le nombre d'états construits de façon importante, mais cela au détriment du temps. Une partie de cette perte de temps est due à la façon dont ces ensembles d'états sont représentés. Comme cette technique demande une représentation ensembliste qui permette d'effectuer des tests d'inclusion et des décompositions rapidement, nous espérons qu'une utilisation des BDD ou des DDD [127] pourrait améliorer ces performances.

Cette application, améliorant une technique existante, ne doit cependant pas occulter le fait que ces algorithmes ont été développés dans un cade générique: toute autre technique pouvant construire un automate satisfaisant les 5 contraintes de la définition 38 pourrait être utilisée.

Nous pensons qu'il devrait être possible de relâcher ces contraintes pour supporter davantage de méthodes, telles que les dépliages [38] ou les graphes d'observation [68], déjà citées, qui construisent toutes les deux des automates ensemblistes ne satisfaisant pas exactement nos conditions.

Chapitre 7

Hypothèses d'équité

Ce chapitre introduit une variante de l'approche automate de la vérification qui prend en compte des *hypothèses d'équité*. Ces hypothèses peuvent être vues comme des filtres appliqués aux exécutions dérivées du modèle: le processus de vérification devra ignorer certaines exécutions *non équitables*. Nous discuterons de deux types d'hypothèses: nous montrons que l'*équité faible* est supportée sans surcoût par certains des algorithmes introduits dans les chapitres précédents, alors que les hypothèses d'*équité forte* ont intérêt à être prises en compte différemment. Pour représenter ces dernières nous utiliserons les automates de Streett pour lesquels nous proposerons un algorithme d'*emptiness check* dédié.

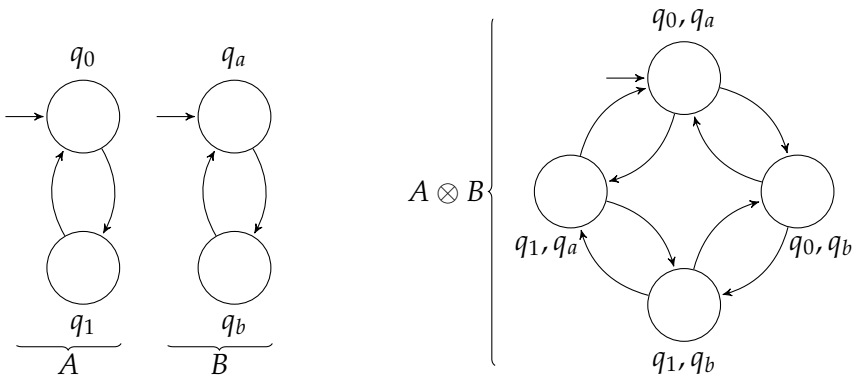
7.1 Introduction

L'*équité* est une notion fréquemment rencontrée lors de l'étude de systèmes concurrents.

Considérons les automates A et B de la figure 7.1a page suivante, dont les étiquettes, disjointes mais autrement sans importance, ne sont pas représentées. Un modèle représentant l'exécution concurrente (avec une sémantique d'entrelacement) de ces deux systèmes est représenté par la figure 7.1b. On voit que dans certaines des exécutions de ce dernier modèle, l'un des deux sous-systèmes reste toujours dans la même configuration. C'est par exemple le cas pour l'exécution qui boucle toujours entre les deux états q_0, q_a et q_1, q_a (le système B reste dans l'état q_a), ainsi que pour celle qui boucle toujours entre q_0, q_a et q_0, q_b (le système A reste dans l'état q_0).

Il est fort possible que l'on sache que de telles exécutions, où l'un des processus reste bloqué, sont en réalité impossibles. Par exemple A et B peuvent être des processus tournant sur une même machine dont on sait que le séquenceur est équitable : c'est-à-dire qu'il donnera la main à chaque processus infiniment souvent. Dans ce cas, les deux exécutions cités ne devraient pas faire partie du langage du modèle afin de ne pas être considérées lors de la vérification de propriétés temporelles.

Plutôt que d'inclure une modélisation du séquenceur de la machine au sein de notre modèle afin de *supprimer* les comportements non-réalistes, nous allons modifier légèrement les procédures de vérification de façon à prendre en compte des hypothèses d'équité



(a) (b)

FIG. 7.1: Deux modèles indépendants A et B, et le modèle $A \otimes B$ résultant de leur mise en concurrence.

et *ignorer* ces comportements. Dans cet exemple, l'hypothèse d'équité serait que chaque processus progresse infiniment souvent.

De telles hypothèses ne portent pas forcément sur des processus. De façon générale, une hypothèse d'équité peut être faite chaque fois que l'on doit répéter un choix entre plusieurs alternatives, dans ce cas, aucun choix ne doit être ignoré pour toujours.

– Dans un contexte de communication via des canaux non fiables, il est fréquent de supposer qu'un message arrivera après un nombre fini d'envois (le *choix répété* est ici de perdre ou de ne pas perdre le message).
– Dans un contexte d'allocation de ressources[1], ce problème est celui de l'*absence de famine*: le prochain utilisateur doit être choisi de façon à ce que personne ne soit laissé pour compte.
– Un autre aspect est celui des attentes finies. Par exemple dans un système où plusieurs processus peuvent accéder à une région critique (où toute autre ressource) on peut vouloir supposer qu'aucun processus ne va se bloquer pour toujours (dans l'attente de la ressource) tandis que les autres processus avancent.

Il faut noter que ces deux dernier points ne sont pas strictement équivalents. Le dernier vise à ce qu'une ressource réclamée de façon continue finisse par être obtenue, tandis que le second cherche à ce qu'une ressource demandée infiniment souvent (mais sans notion de blocage et d'attente) soit obtenue. La seconde hypothèse implique la dernière, mais pas l'inverse.

7.2 Définitions

Les hypothèses d'équité pouvant s'appliquer à différents niveaux d'abstraction, nous définirons les différentes hypothèses en parlant d'événements. Dans ces travaux, un événe-

[1] Le séquenceur équitable est un cas particulier d'allocation de ressources: les ressources étant les quantums de temps d'utilisation du processeur.

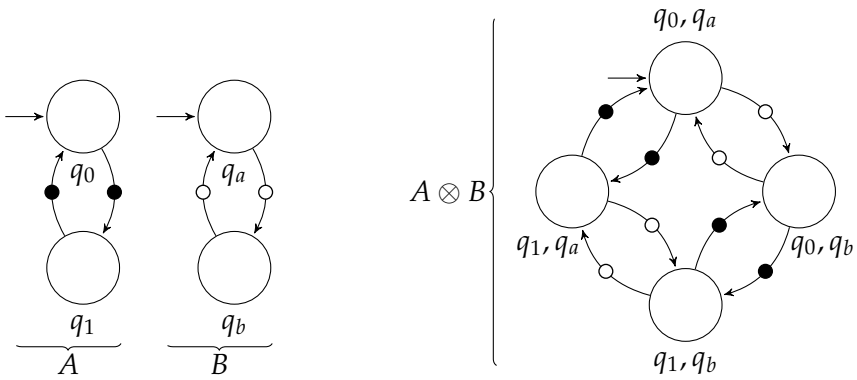


FIG. 7.2: Équité inconditionnelle en utilisant des TGBA pour représenter le système de la figure 7.1 page précédente.

ment peut être aussi bien l'avancée d'un processus (quelle que soit la transition franchie) lorsqu'on parle de processus équitable, que le franchissement d'une transition particulière si un choix particulier doit être équitable.

Parmi les différentes hypothèse d'équité existantes [52], nous nous cantonnerons aux trois notions suivantes (que nous réduirons rapidement à deux):
– équité inconditionnelle
– équité faible
– équité forte

Définition 44. *Une* hypothèse d'équité inconditionnelle *spécifie qu'un événement se produira infiniment souvent.*

On peut ignorer les exécutions où l'un des deux processus de la figure 7.1b page précédente n'avance pas en faisant deux hypothèses d'équité inconditionnelle: on suppose que chaque processus progresse infiniment souvent.

Ce type d'équité peut être pris en compte en remplaçant les structures de Kripke de la figure 7.1 page ci-contre par des TGBA dans lesquelles une même condition d'acceptation est associée à chaque transition d'un processus comme le montre la figure 7.2. Le fait que les exécutions acceptantes de l'automate $A \otimes B$ passent obligatoirement infiniment souvent par des transitions étiquetées par «●» et «○» assure qu'aucun des deux processus ne sera infiniment ignoré par le séquenceur.

Comme on l'a dit, ce type de condition d'acceptation peut aussi s'appliquer à une transition particulière plutôt qu'à un processus entier. Il suffit alors de créer une nouvelle condition d'acceptation pour étiqueter cette transition.

L'équité inconditionnelle peut aussi s'exprimer comme une formule LTL de la forme

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

où e est une propriété étiquetant le système et qui doit être vraie si et seulement si l'événement considéré se produit. Cela permet, si on le souhaite, d'avoir une autre approche de la vérification sous hypothèse d'équité. Vérifier une propriété LTL φ sous une hypothèse d'équité exprimée par la formule ψ, revient à vérifier la formule $\psi \rightarrow \varphi$.

Si nous notons S l'automate représentant le système à vérifier, nous voyons que

$$\begin{aligned} S \otimes A_{\neg(\psi \rightarrow \varphi)} &= S \otimes A_{\neg(\neg\psi \vee \varphi)} \\ &= S \otimes A_{\psi \wedge \neg\varphi} \\ &= S \otimes A_{\psi} \otimes A_{\neg\varphi} \end{aligned} \tag{7.1}$$

Nous pouvons conclure deux choses de cette dernière formule. D'une part, vérifier la formule $\psi \rightarrow \varphi$ sur le système S est donc équivalant à vérifier la formule φ sur le système $S \otimes A_{\psi}$. Intégrer l'hypothèse d'équité au système à vérifier est ce que nous avions fait dans l'exemple de la figure 7.2 en ajoutant des conditions d'acceptation.

D'autre part, $S \otimes \mathsf{A}_{\neg\varphi}$ correspondant à la vérification sans hypothèse d'équité, la complexité ajoutée par la prise en compte de cette hypothèse est directement liée à l'automate A_{ψ}.

Dans le cas général où nous faisons n hypothèses d'équité inconditionnelle, la formule $\psi = \bigwedge_{i=1}^{n} \mathsf{GF}\, e_i$ peut être traduite en un TGBA A_{ψ} déterministe possédant un unique état et n conditions d'acceptation. La figure 3.11 page 57 illustre le cas $n = 2$. Le produit par A_{ψ} ne fera donc pas croître la taille de l'automate à explorer par l'*emptiness check*, mais augmentera simplement le nombre de conditions d'acceptation.

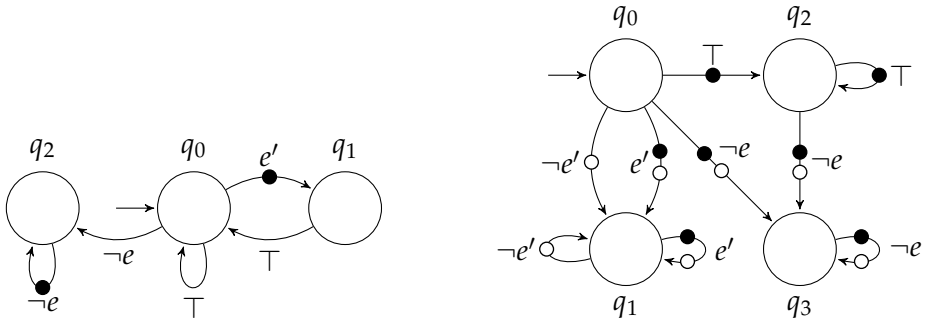
Comme les *emptiness checks* basés sur les CFC sont insensibles au nombre de conditions d'acceptation, nous pouvons en déduire que les hypothèses d'équité inconditionnelle ont un surcoût nul pour la vérification par ce type d'algorithme. Le choix de faire porter les conditions d'acceptation sur les transitions est particulièrement important dans ce cas ; si elles avaient été sur les états le nombre d'hypothèses d'équité aurait conduit à une explosion du nombre d'états. (Voir par exemple la propriété 4 page 51, et comparer les figures 3.11 page 57 et 12 page 45.)

Définition 45. *Une* hypothèse d'équité faible *spécifie que, si un événement se produit de façon continue, un autre événement se produira infiniment souvent.*

Si l'on note e et e' ces événements, cette hypothèse peut s'exprimer de la façon suivante en LTL:

$$\mathsf{FG}\, e \rightarrow \mathsf{GF}\, e'$$

Notre définition est un peu plus générale que les définitions d'équité faible rencontrées dans la littérature [52, 89, 88] où elle est exprimée en termes de franchissabilité et d'occurence. Un événement est faiblement équitable s'il se produit infiniment souvent lorsqu'il peut se produire de façon continue. Ainsi un processus qui peut prendre une ressource de façon continue finira par la prendre. Nous considérons, pour notre part, la *possibilité* de l'événement comme un événement en lui-même.



(a) (b)

FIG. 7.3: Deux TGBA acceptant $\mathsf{GF}\,e \rightarrow \mathsf{GF}\,e'$.

L'expression de l'équité faible sous la forme d'une formule LTL nous permet de voir aisément comment elle se ramène à une formule d'équité inconditionnelle:

$$\begin{aligned}\mathsf{FG}\,e \rightarrow \mathsf{GF}\,e' &= \neg\,\mathsf{FG}\,e \vee \mathsf{GF}\,e' \\ &= \mathsf{GF}\,\neg e \vee \mathsf{GF}\,e' \\ &= \mathsf{GF}(\neg e \vee e')\end{aligned}$$

Traiter l'équité faible est donc aussi simple que traiter l'équité inconditionnelle. Aussi, en dehors de cette section nous ne distinguerons pas équités inconditionnelle et faible, et considérerons que l'équité inconditionnelle n'est qu'un cas particulier de l'équité faible.

Définition 46. *Une* hypothèse d'équité forte *spécifie que si un événement se produit infiniment souvent, un autre événement se produira infiniment souvent.*

C'est le cas des communications avec pertes de messages. On suppose que si un message est envoyé infiniment souvent, il arrivera infiniment souvent (autrement dit: on ne les perd pas tous).

En notant toujours e et e' ces événements, cette hypothèse peut s'exprimer par:

$$\mathsf{GF}\,e \rightarrow \mathsf{GF}\,e'$$

Cette fois-ci, la formule ne se réduit pas aux précédentes. La réécriture sous la forme $(\mathsf{FG}\,\neg e) \vee (\mathsf{GF}\,e')$ permet de mieux comprendre les traductions sous forme de TGBA proposées par la figure 7.3.

L'automate de la figure 7.3b a été obtenu par l'algorithme de traduction de la section 4.2 page 86. Une seconde passe sur cet automate permettrait de simplifier ses conditions d'acceptation. Les seuls chemins acceptants sont ceux qui finissent par passer infiniment souvent par les boucles à droite des états q_1 et q_3: on peut donc supprimer toutes les conditions d'acceptation et ne laisser qu'un « $\bullet$ » sur chacune de ces deux boucles.

L'automate de la figure 7.3a est une adaptation sur les transitions d'un automate de Büchi étiqueté sur les états proposé par Kesten et al. [84]. Il n'existe pas, à notre connaissance,

d'algorithme de traduction de formule LTL produisant un tel automate. Cependant la différence de taille entre ces deux automates n'est pas très significative. Lorsqu'on synchronise plusieurs de ces automates, on peut estimer que c'est le nombre d'états faisant partie d'une composante fortement connexe qui va influer sur la taille du résultat. Dans ces deux automates ce nombre est le même: 3. On comprend que chercher à exprimer la formule

$$\bigwedge_{i=1}^{n} \mathsf{GF}\, e_i \rightarrow \mathsf{GF}\, e'_i$$

en synchronisant des automates comme celui de la figure 7.3a produira un automate possédant 3^n états. De façon empirique, on constate que traduire cette formule avec l'algorithme de la section 4.2 page 86 produit des automates à $3^n + 1$ états.

D'après l'équation 7.1 page 162, nous en déduisons que la vérification sous n hypothèses d'équité forte avec des TGBA (et en appliquant des *emptiness checks* basés sur les CFC) provoque un surcoût exponentiel en n par rapport à la vérification sans ces hypothèses.

7.3 Automates de Streett

Nous introduisons maintenant un autre type d'ω-automates, mieux adapté à la prise en compte d'hypothèses d'équité forte.

Définition 47. *Un* automate de Streett étiqueté sur les transitions *est un quintuplet* $A = \langle AP, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta \rangle$ *où*
- AP *est un ensemble fini de propositions atomiques,*
- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $\mathcal{Q}^0 \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
- $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_r, u_r)\}$ *est un ensemble fini de paires d'éléments appelés conditions d'acceptation et supposés tous différents,*
- $\delta \subseteq \mathcal{Q} \times 2^{2^{AP}} \times 2^{\{l_1,\ldots,l_r,u_1,\ldots,u_r\}} \times \mathcal{Q}$ *est la relation de transition de l'automate (chaque transition étant étiquetée par une formule propositionnelle ainsi qu'un ensemble de conditions d'acceptation).*

La différence entre les automates Streett et les TGBA tient uniquement à l'interprétation des conditions d'acceptation, aussi nous réutiliserons les notations de la section 3.3.4 page 48. Notamment, $\mathrm{Run}(A)$ désigne l'ensemble des chemins infinis parcourant l'automate à partir d'un état de $\mathcal{Q}^0$, et pour toute transition $t \in \delta$, nous noterons $t = (t^{\mathrm{in}}, t^{\mathrm{prop}}, t^{\mathrm{acc}}, t^{\mathrm{out}})$.

Définition 48. *Les chemins acceptants d'un automate de Streett étiqueté sur les états* $\langle AP, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta \rangle$ *sont les chemins infinis qui, pour chaque paire de conditions* $(l_i, u_i) \in \mathcal{F}$*, traversent infiniment souvent des transitions étiquetées par* u_i *s'ils traversent infiniment souvent des transitions étiquetées par* l_i*.*

$$\mathrm{Acc}(A) = \{r \in \mathrm{Run}(A) \mid \forall (u,l) \in \mathcal{F},\ \left(\forall i \geq 0,\ \exists j \geq i,\ l \in r(j)^{\mathrm{acc}}\right) \implies \left(\forall i \geq 0,\ \exists j \geq i,\ u \in r(j)^{\mathrm{acc}}\right)\}$$

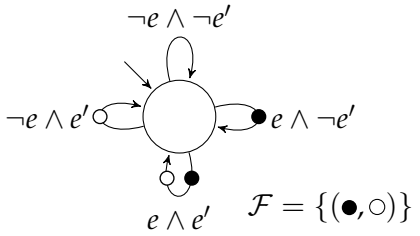


FIG. 7.4: Automate de Streett étiqueté sur les transitions et acceptant $\mathsf{G}\,\mathsf{F}\,e \rightarrow \mathsf{G}\,\mathsf{F}\,e'$.

Le langage associé à l'automate est toujours l'ensemble des exécutions pour lesquelles il existe un chemin acceptant:

$$\mathscr{L}(A) = \{\sigma \in \Sigma^\omega \mid \exists t_0 \cdot t_1 \cdot t_2 \cdots \in \mathrm{Acc}(A),\ \forall i \in \mathbb{N},\ \sigma(i) \in t_i^{\mathrm{prop}}\}$$

Les automates de Streett sont aussi expressifs que les TGBA.

Les TGBA peuvent être vus comme des cas particuliers d'automates de Streett où les u_i correspondent aux conditions d'acceptation du TGBA, et les l_i étiquettent toutes les transitions de l'automate. Plus formellement, le TGBA $A_T = \langle AP, \mathcal{Q}, \mathcal{Q}^0, \{u_1, u_2, \ldots, u_r\}, \delta\rangle$ possède le même langage que l'automate de Streett étiqueté sur les transitions $A_S = \langle AP, \mathcal{Q}, \mathcal{Q}^0, \{(l_1, u_1), (l_2, u_2), \ldots, (l_r, u_r)\}, \delta'\rangle$ où

$$\delta' = \{\langle t^{\mathrm{in}}, t^{\mathrm{prop}}, t^{\mathrm{acc}} \cup \{l_1, l_2, \ldots, l_r\}, t^{\mathrm{out}}\rangle \mid t \in \delta\}.$$

En revanche, il n'existe pas d'algorithme de complexité polynomiale pour traduire un automate de Streett en un automate de Büchi (quel qu'il soit) [104, chapitre 4].

La figure 7.4 montre un automate de Streett étiqueté sur les transitions traduisant une hypothèse d'équité forte. Cet automate est déterministe et ne possède qu'un état.

De façon générale pour n hypothèses d'équité forte:

$$\bigwedge_{i=1}^{n} \mathsf{G}\,\mathsf{F}\,e_i \rightarrow \mathsf{G}\,\mathsf{F}\,e'_i$$

nous pouvons construire un automate de Streett étiqueté sur les transitions, déterministe, possédant un seul état, 2^{2n} transitions et n paires de conditions d'acceptation. Il s'agit de l'automate $A = \langle AP, \{q\}, \{q\}, \mathcal{F}, \delta\rangle$ avec
– $AP = \{e_1, e_2, \ldots, e_n, e'_1, e'_2, \ldots e'_n\}$,
– $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$ et
– $\delta = \{\langle q, \{E\}, \alpha(E), q\rangle \mid E \in 2^{AP}\}$ où $\alpha(E) = \{l_i \mid e_i \in E\} \cup \{u_i \mid e'_i \in E\}$.
Si un tel automate est utilisé à la place de A_ψ dans l'équation 7.1 page 162, nous en déduisons que dans le cadre d'une vérification s'appuyant sur les automates de Streett, l'ajout d'n hypothèses d'équité forte n'augmente pas la taille de l'automate à tester: cela n'ajoute que n paires de conditions d'acceptation. En pratique, le produit synchronisé[2] avec A_ψ

[2]Le produit synchronisé de deux automates de Streett se définit de façon similaire à celui de deux TGBA donné par la définition 31 page 53, seule la façon de renommer les éléments de $\mathcal{F}$ doit changer: dans les automates de Streett nous renommerons les l_i et u_i de chaque automate pour qu'ils soient distincts.

est inutile: la fonction α ci-dessus peut être utilisée directement lors de la génération de l'espace d'état pour étiqueter les transitions avec les bonnes conditions d'acceptation. Naturellement cela demande un autre algorithme d'*emptiness check*.

Dans la section 7.6 page 168 nous présenterons un algorithme d'*emptiness check* pour les automates de Streett qui, s'il partage la même classe de complexité que celui pour les TGBA (PSPACE), est quand même linéairement plus lent par rapport au nombre de paires de conditions d'acceptation.

7.4 Cas pratique

Sebastiani et al. [110] citent la propriété LTL suivante comme particulièrement difficile à vérifier.

$$\begin{aligned}\Big(&(\mathsf{GF}\,p_0 \to \mathsf{GF}\,p_1) \wedge (\mathsf{GF}\,p_2 \to \mathsf{GF}\,p_0)\wedge\\ &(\mathsf{GF}\,p_3 \to \mathsf{GF}\,p_2) \wedge (\mathsf{GF}\,p_4 \to \mathsf{GF}\,p_2)\wedge\\ &(\mathsf{GF}\,p_5 \to \mathsf{GF}\,p_3) \wedge (\mathsf{GF}\,p_6 \to \mathsf{GF}(p_5 \vee p_4))\wedge\\ &(\mathsf{GF}\,p_7 \to \mathsf{GF}\,p_6) \wedge (\mathsf{GF}\,p_1 \to \mathsf{GF}\,p_7)\Big) \to \mathsf{GF}\,p_8\end{aligned} \tag{7.2}$$

La traduction de sa négation en automate de Büchi ferait exploser la plupart des algorithmes. Dans leurs tests, la seule implémentation qui soit venue à bout de cette traduction serait le traducteur de Spot mais encore aurait-elle produit un automate de plus 70000 états.

Comme nous l'avons montré section 4.4 page 98, Spot implémente deux algorithmes de traduction d'une formule LTL en automate de Büchi, le premier (Spot/LaCIM), peu efficace, est une construction basée sur les BDD [35], tandis que le second (Spot/FM) implémente les idées développées dans les sections 4.2 à 3.3.7 pages 86–57. Il est fort probable que c'est la première de ces constructions qui a donné un si gros automate. La seconde produit un automate de 7291 états sans optimisation, ou 1731 états en activant toutes les optimisations.

Avec un algorithme dédié Sebastiani et al. parviennent à traduire cette formule en un automate de 1281 états, ce qui reste tout de même assez important pour un automate exprimant une propriété LTL.

Si nous étudions la formule 7.2, nous pouvons voir qu'elle est de la forme $\psi \to \varphi$ où la formule ψ est un regroupement d'hypothèses d'équité forte. Nous sommes donc exactement dans le cas de l'équation 7.1 page 162. L'automate A_ψ peut être représenté par un automate de Streett déterministe à un seul état (page 165) et 8 paires de conditions d'acceptation. D'autre part, l'automate $A_{\neg\varphi} = A_{\neg\mathsf{GF}\,p_8} = A_{\mathsf{FG}\neg p_8}$ peut être représenté par un automate de Büchi non-deterministe à deux états et une condition d'acceptation. La négation de la formule 7.2 peut donc être représentée par un automate de Streett avec 2 états et 9 paires de conditions d'acceptation.

7.5 Équité pour les systèmes de transitions synchronisés

Dans la section 2.6 page 27, nous avions expliqué que soumettre les règles (5) et (7) de la figure 2.2 page 18 à des hypothèses d'équité forte suffisait à supprimer les comportements du modèle dans lesquels l'un des messages n'est jamais reçu.

Nous avons d'autre part expliqué que les hypothèses d'équité avaient intérêt a être prise en compte dans l'automate représentant l'espace d'état, et non sous la forme d'une formule LTL séparée.

Dans ce contexte, l'automate représentant l'espace des états accessibles du modèle n'est plus une structure de Kripke (2.4 page 21) mais un automate de Streett. Cet automate doit être étiqueté par des conditions d'acceptation de façon à respecter les contraintes de ces hypothèses.

Nous souhaitons indiquer ici la façon d'étiqueter l'automate de Streett pour diverses hypothèses. Lorsque l'hypothèse porte sur une règle de synchronisation, les e_i et e'_i correspondent respectivement aux propriétés « la règle pouvait être appliquée dans l'état source » (activabilité) et « la transition courante correspond à la règle » (occurrence ou activation).

Règle de synchronisation avec équité forte. Pour toute règle de synchronisation avec équité forte, nous créons une nouvelle paire de conditions d'acceptation (l_i, u_i). Les transitions correspondant à des instances de la règle sont étiquetées par u_i. Toutes les transitions quittant le même état source qu'une transition étiquetées par u_i doivent être étiquetée par l_i.

De cette façon, nous garantissons que chaque fois que l'automate *aurait pu* suivre une transition étiquetée par u_i, il traverse une transition étiquetée par l_i. Ainsi les conditions d'acceptation de Streett nous assurent que si la règle est activable infiniment souvent, des transitions correspondantes seront franchies infiniment souvent.

Règle de synchronisation avec équité faible. Pour toute règle de synchronisation avec équité faible, nous créons une nouvelle paire de conditions d'acceptation (l_i, u_i). Absolument toutes les transitions sont étiquetées par l_i (car nous souhaitons simuler des conditions d'acceptation de Büchi). Les transitions correspondant à des instances de la règle sont étiquetées par u_i (un chemin est acceptant si la règle est activée infiniment souvent). Les transitions qui ne partagent pas d'état source avec une instance de la règle sont aussi étiquetées par u_i (un chemin est aussi acceptant si la règle n'est pas activable infiniment souvent).

De cette façon, nous évitons les chemins où la règle est activable continûment sans jamais être activée.

Les conditions de cet étiquetage peuvent être évaluées localement lors du calcul de l'ensemble des successeurs de chaque état ; la construction à la volée de l'espace d'état n'est donc pas remise en question.

Nous pouvons aussi imaginer des hypothèses d'équité qui portent sur un système de transition dans son ensemble. Par exemple il est classique de supposer que si deux processus tournent sur une même machine, ils progresseront tous les deux s'ils ne sont pas

bloqués (c'est-à-dire que l'un des processus ne s'accapare le processeur). Cette hypothèse d'équité faible peut se traduire de la façon suivante: pour chaque processus nous créons un paire de conditions d'acceptation (l_i, u_i), toutes les transitions sont étiquetées par l_i, et nous étiquetons par u_i les transitions où le système de transitions du processus en question change d'état (le processus progresse), ainsi que celles dont l'état source ne permet pas au processus de changer d'état (il est bloqué).

7.6 Emptiness Check d'automates de Streett

L'algorithme que nous proposons pour répondre au problème de l'emptiness check d'un automate de Streett est dérivé de celui pour les TGBA développé par Couvreur [34] et présenté section 5.4.1 page 120.

Son fonctionnement, décrit ci-après, est illustré sur un exemple par la figure 7.5 page 170. L'automate dont on cherche à tester la vacuité est un automate de Streett à 5 états et 2 paires de conditions d'acceptation: (●, ○) et (●, ●).

Comme dans le cas des TGBA, l'algorithme effectue une recherche en profondeur pour marquer les composantes fortement connexes rencontrées. Chaque CFC non-triviale est étiquetée par la liste des conditions d'acceptation qui s'y trouvent. Les figures 7.5a–7.5f montrent ces premières étapes. Lorsqu'une CFC terminale est atteinte nous pouvons nous trouver dans l'une des trois situations qui suivent. Notons $\mathcal{F} = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\}$ l'ensemble des conditions d'acceptation de l'automate de Streett et acc l'ensemble des conditions d'acceptation de la CFC terminale.

1. Soit la CFC est triviale (aucune boucle): elle ne peut donc être acceptante et doit être retirée comme c'était le cas dans l'*emptiness check* pour TGBA.
2. Soit la composante est acceptante: $\forall i, l_i \in acc \implies u_i \in acc$.
   Dans ce cas l'algorithme termine en indiquant l'existence d'un chemin acceptant. Comme dans l'*emptiness check* pour TGBA, ce test a tout intérêt à être effectué chaque fois qu'une CFC non triviale est formée et pas uniquement pour les CFC terminales. Cela permet d'interrompre l'algorithme plus tôt.
3. Soit $\exists i, l_i \in acc \wedge u_i \notin acc$.
   Dans ce cas nous ne pouvons pas dire immédiatement si la composante est acceptante ou non. Il est peut-être possible d'y trouver un circuit acceptant ne passant par aucune transition étiquetée par l_i.

La figure 7.5f illustre ce troisième cas: $\mathcal{F} = \{(●, ○), (●, ●)\}$ et $acc = \{●, ●, ●\}$ donc l'algorithme ne peut pas conclure immédiatement.

Lorsque l'algorithme se trouve dans cette situation, nous proposons de parcourir à nouveau la CFC, en évitant les transitions problématiques t telles que $\exists i, l_i \in t^{\text{acc}} \wedge u_i \notin acc$. Concrètement, l'algorithme définit l'ensemble $avoid = \{l_i \in acc \mid u_i \notin acc\}$ des l_i qui ne peuvent être satisfaits. Les états de la CFC sont marqués comme non-visités, et l'algorithme parcourt la composante à nouveau avec les changements suivants :
– parmi les transitions sortantes d'un état, celles qui portent des conditions d'acceptation de *avoid* sont visitées en dernier

– le franchissement d'une transition portant une condition d'acceptation de *avoid* provoque la pose d'une « barrière », symbolisée par des pointillés verticaux sur la figure 7.5i
– une transition reliant deux CFC ne permet la fusion de ces CFC que si elles sont après la dernière barrière posée: le passage de la figure 7.5i à la 7.5j est un cas où la fusion est possible, alors que les CFC de la figure 7.5k ne peuvent pas être fusionnées.

Cette méthode va donc construire de plus petites CFC à la place de la CFC originale. La seule façon de fusionner ces CFC serait de prendre en compte un circuit étiqueté par des conditions d'acceptation (de *avoid*) qui ne peuvent être satisfaites. On peut alors déterminer pour chacune de ces CFC si elle est triviale, acceptante, ou encore si elle contient des conditions d'acceptation (non listées dans *avoid*) non satisfaites. Dans ce dernier cas on itère à nouveau le processus: compléter *avoid* et reparcourir la CFC en posant des barrières. Cette procédure récursive termine car $|avoid|$ croît à chaque itération et ne peut pas dépasser $|\mathcal{F}|$.

Par rapport à l'algorithme pour les TGBA, qui ne visitait chaque état qu'une seule fois, on voit que celui-ci peut, dans le pire des cas, visiter chaque état $|\mathcal{F}| + 1$ fois.

Notons enfin que le premier parcours découvrant la composante (sans éviter de condition d'acceptation) peut être vu comme un cas particulier des parcours suivants avec $avoid = \emptyset$.

L'idée de reparcourir la CFC en ignorant les transitions qui gênent la satisfaction n'est pas nouvelle, c'est notre mise en œuvre qui l'est. Les algorithmes existants pour l'*emptiness check* d'automates de Streett (étiquetés sur les états) [102, 90] utilisent aussi une décomposition des automates en composantes fortement connexes. Pour chaque CFC qui n'est pas acceptante, c'est-à-dire pour laquelle $\exists i, l_i \in acc \wedge u_i \notin acc$, ils retirent explicitement tous les états qui ne sont étiquetés par l_i avant de faire une nouvelle passe sur l'automate ainsi amputé. Ce procédé se prête mal à un calcul à la volée car il demande de manipuler le graphe de l'automate explicitement.

La figure 7.6 page 172 donne une implémentation possible de cet algorithme. L'algorithme reprend une grande part de la structure de celui de la figure 5.8 page 121. Comme dans la seconde heuristique discutée section 5.4.2 page 127, le parcours en profondeur est ici effectué en terme de CFC et non d'états. La pile *todo* du parcours en profondeur de l'*emptiness check* pour les TGBA a donc été ici fusionnée avec la pile *SCC* des composantes fortement connexes.

Les successeurs non explorés d'une composante fortement connexe sont partionnés en deux ensembles *succ* et $fsucc$ qui permettent de fixer l'ordre de franchissement des transitions d'un état: lorsqu'un état est empilé sur *SCC* par la ligne 29, $fsucc$ reçoit toutes les transitions étiquetées par une condition d'acceptation qui doit être évitée, et *succ* reçoit les autres. Ces dernières seront visitées en priorité: l'algorithme prend toujours le prochain successeur dans *succ* (ligne 17) et ne permute $fsucc$ et *succ* que lorsque ce dernier est vide (lignes 10–12).

Les « barrières », censées empêcher la fusion des composantes fortement connexes qui utiliseraient une transition étiquetée par une condition d'acceptation ne pouvant être satisfaite, sont représentées par le numéro du dernier état de la CFC source de la transition. Cette CFC est forcément celle du sommet de la pile, et le numéro du dernier état est max.

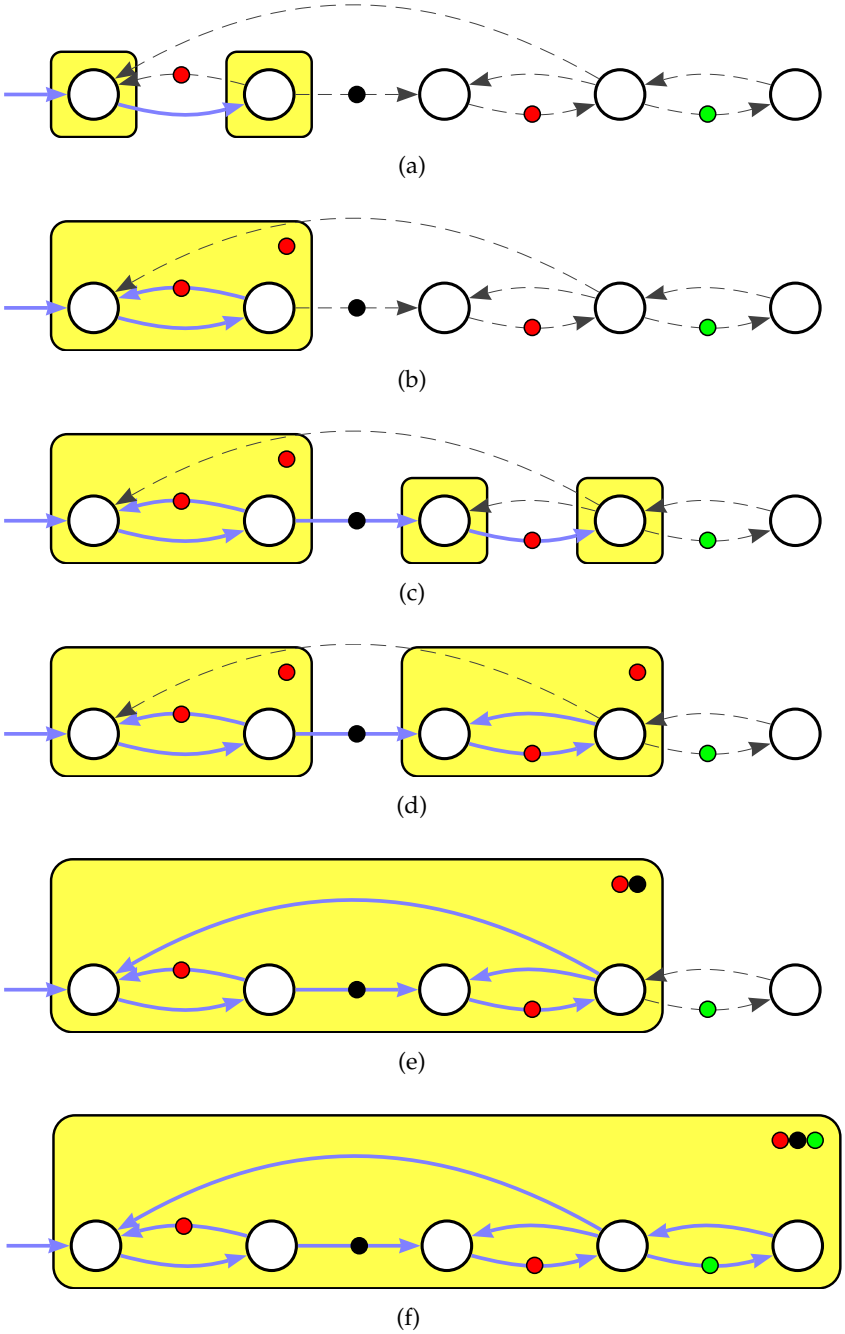


FIG. 7.5: *Emptiness check* d'un automate de Streett avec $\mathcal{F} = \{(\bullet, \circ), (\bullet, \bullet)\}$.

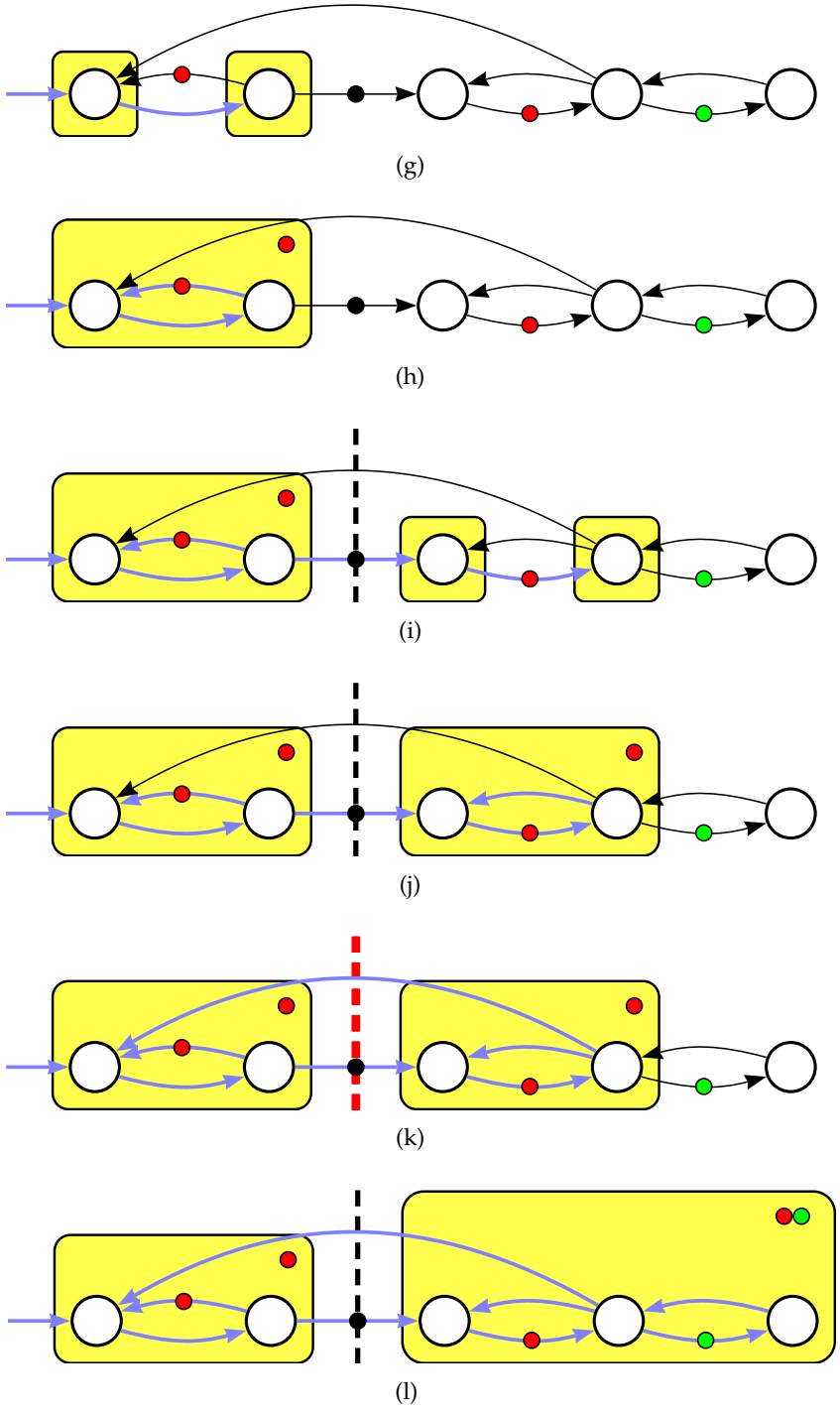


FIG. 7.5: Suite.

```
1  Entrée : Un automate de Streett A = ⟨AP, Q, Q^0, F, δ⟩
2  Résultat : ⊤ si et seulement si L(A) = ∅
3  Données :  SCC: stack of
                    ⟨state ∈ Q, root ∈ ℕ, la ⊆ F, acc ⊆ F, rem ⊆ Q, succ ⊆ δ, fsucc ⊆ δ⟩
              H: map of Q ↦ ℕ
              avoid: stack of ⟨root ∈ ℕ, acc ⊆ F⟩
              min: stack of ℕ
              max ← 0
4  begin
5      min.push(0)
6      avoid.push(⟨1, ∅⟩)
7      forall q^0 ∈ Q^0 do
8          DFSpush(∅, q^0)
9          while ¬SCC.empty() do
10             if SCC.top().succ = ∅ then
11                 if SCC.top().fsucc ≠ ∅ then
12                     swap(SCC.top().succ, SCC.top().fsucc)
13                     min.push(max)
14                 else
15                     DFSpop()
16             else
17                 pick one ⟨s, _, a, d⟩ off SCC.top().succ
18                 if d ∉ H then
19                     DFSpush(a, d)
20                 else if H[d] > min.top() then
21                     merge(a, H[d])
22                     acc ← SCC.top().acc
23                     if ∀⟨l, u⟩ ∈ F, (l ∈ acc) ⟹ (u ∈ acc) then return ⊥
24     return ⊤
25 end
```

FIG. 7.6: Algorithme d'*emptiness check* pour automates de Streett.

```
26 DFSpush(a ⊆ F, q ∈ Q)
27     max ← max + 1
28     H[q] ← max
29     SCC.push(⟨q, max, a, ∅, ∅, {⟨s, l, a, d⟩ ∈ δ | s = q, a ∩ avoid.top().acc = ∅},
                                  {⟨s, l, a, d⟩ ∈ δ | s = q, a ∩ avoid.top().acc ≠ ∅}⟩)
30 end
31 DFSpop()
32     ⟨q, n, la, acc, rem, _, _⟩ ← SCC.pop()
33     max ← n − 1
34     if n ≤ min.top() then
35         min.pop()
36     old_avoid ← avoid.top().acc
37     if n = avoid.top().root then
38         avoid.pop()
39     new_avoid ← old_avoid ∪ {l | ⟨l, u⟩ ∈ F, l ∩ acc ≠ ∅, u ∩ acc = ∅}
40     if new_avoid ≠ old_avoid then
41         foreach s ∈ rem do
42             delete H[s]
43         avoid.push(n, new_avoid)
44         DFSpush(la, q)
45     else
46         foreach s ∈ rem do
47             H[s] ← 0
48 end
49 merge(a ⊆ F, t ∈ ℕ)
50     r ← ∅
51     s ← ∅
52     f ← ∅
53     while t < SCC.top().root do
54         a ← a ∪ SCC.top().acc ∪ SCC.top().la
55         r ← r ∪ SCC.top().rem ∪ SCC.top().state
56         s ← s ∪ SCC.top().succ
57         f ← f ∪ SCC.top().fsucc
58         SCC.pop()
59     SCC.top().acc ← SCC.top().acc ∪ a
60     SCC.top().rem ← SCC.top().rem ∪ r
61     SCC.top().succ ← SCC.top().succ ∪ s
62     SCC.top().fsucc ← SCC.top().fsucc ∪ f
63 end
```

FIG. 7.6: Suite.

Ces numéros sont empilés sur une pile appelée *min* utilisée ligne 20 pour décider si une transition atteignant une CFC en amont du chemin de recherche doit provoquer la fusion de toutes les CFC intermédiaires. Ces « barrières » sont posées ligne 13 chaque fois que l'algorithme commence à considérer les transitions de $fsucc$ pour un état, et sont retirées ligne 35 quand l'état correspondant est lui-même retiré.

Les conditions d'acceptation à éviter sont stockées en haut d'une pile appelée *avoid*. Celle-ci est complétée chaque fois que l'algorithme a besoin de reparcourir une composante fortement connexe (ligne 43). Chaque élément de cette pile est une paire indiquant le numéro d'état *root* de la racine de la composante fortement connexe à partir de laquelle les conditions d'acceptation *acc* doivent être évitées. L'élément est retiré de la pile lorsque la composante a été entièrement parcourue et qu'elle doit être retirée (lignes 37–38).

Les algorithmes d'*emptiness check* pour TGBA et automate de Streett partageant la même structure, on comprendra que leur preuve est similaire.

Gardons des notations semblables à celles de la page 120 :

$$\begin{aligned}
SCC =& \langle state_0, root_0, la_0, acc_0, rem_0, succ_0, fsucc_0\rangle \\
& \langle state_1, root_1, la_1, acc_1, rem_1, succ_1, fsucc_1\rangle \\
& \vdots \\
& \langle state_n, root_n, la_n, acc_n, rem_n, succ_n, fsucc_n\rangle \\
min =& min_0 min_1 \dots min_p \\
avoid =& \langle ar_0, \overline{acc_0}\rangle \langle ar_1, \overline{acc_1}\rangle \dots \langle ar_r, \overline{acc_r}\rangle \\
\mathfrak{S}_i =& \{s \in \mathcal{Q} \mid root_i \leq H[s] < root_{i+1}\} \qquad \text{pour } 0 \leq i < n \\
\mathfrak{S}_n =& \{s \in \mathcal{Q} \mid root_n \leq H[s]\}
\end{aligned}$$

Lemme 1. *À tout moment de l'exécution de la fonction principale (lignes 9–23), pour toute entrée $\langle ar_i, \overline{acc_i}\rangle$ de la pile avoid il existe une entrée unique $\langle state_j, root_j, la_j, acc_j, rem_j, succ_j, fsucc_j\rangle$ de SCC telle que $ar_i = root_j$. Autrement dit, les entrées de avoid sont toujours associées à des racines des composantes fortement connexes.*

Démonstration. La propriété est vraie initialement après l'initialisation des lignes 6–8. Les manipulations de *avoid* lignes 38 et 43 n'interviennent qu'en même temps que les changements de *SCC* correspondants (lignes 32 et 44 respectivement).

Le danger ne peut donc venir que du seul endroit où l'on supprime des éléments de *SCC* sans consulter *avoid*: la ligne 58 dans la procédure de fusion.

Le scénario qui pourrait poser problème est le suivant. Supposons qu'il existe une entrée $SCC[j] = \langle state_j, root_j, la_j, acc_j, rem_j, succ_j, fsucc_j\rangle$ de la pile *SCC* associée à une entrée $\langle ar_i, \overline{acc_i}\rangle$ de *avoid* (i.e., $root_j = ar_i$). Supposons l'existence d'une composante antérieure $SCC[k]$ avec $k < j$ accessible depuis la composante au sommet de la pile $SCC[n]$. Dans ce cas lorsque l'algorithme trouve la transition entre $SCC[n]$ et $SCC[k]$, il appelle alors `merge()` pour fusionner les composantes $SCC[k]$ à $SCC[n]$ en une seule composante de racine $root_k < ar_i$ et $\langle ar_i, \overline{acc_i}\rangle$ ne correspond plus à une racine de CFC.

Ce scénario ne peut se produire. En effet, les entrées de *avoid* sont ajoutées pour des racines de CFC par `DFSpop()`, c'est-à-dire seulement une fois que tous les successeurs de l'état ont été visités: si la transition entre $SCC[n]$ et $SCC[k]$ existait, les CFC auraient été fusionnées lors de ce parcours et $\langle ar_i, \overline{acc_i} \rangle$ n'aurait pu être associé à $SCC[j]$ (qui n'existe plus). □

Lemme 2. *Au moment où la ligne 17 est exécutée pour choisir un état parmi les successeurs du sommet de SCC, la valeur de $\overline{acc_r}$ est la même que lorsque cet ensemble de successeurs avait été créé ligne 29.*

Démonstration. Les valeurs des éléments empilés sur *avoid* ne changent pas. La taille, r, de la pile (et donc la valeur de $\overline{acc_r}$) ne peut changer que lignes 38 et 43.

Si une entrée est ajoutée dans la pile *avoid* ligne 43, la fonction `DFSpush` est aussi appelée ligne 44, augmentant n.

Entre un empilage ligne 17 et le choix d'un successeur ligne 29 avec la même valeur de n, il y aura forcément eu une séquence de `DFSpush` et de `DFSpop` ayant ramené n à cette même valeur.

Il ne peut y avoir qu'une entrée dans *avoid* pour un n donné (d'après le lemme 1) et chaque appel à `DFSpop` teste si cette entrée doit être retirée. Ceci garantit que le sommet de *avoid* lorsque la ligne 17 est exécutée était le même lorsque la ligne 29 a empilé cet ensemble de successeurs. □

Lemme 3. *Les valeurs de $(root_i)_{i \in [\![0,n]\!]}$ sont strictement croissantes et l'on a toujours $root_n \leq max$ pour toutes les lignes de la fonction principale (9–23).*

Démonstration. La propriété est vraie initialement. Les seules lignes pouvant l'affecter sont les lignes 27–29, 32–33 et 58, mais il est clair que toutes ces modifications sont correctes. □

Lemme 4. *Notons n' la valeur de n à un moment où les lignes 12–13 sont exécutées. Les ensembles $succ_{n'}$ et $fsucc_{n'}$ ne croîtrons plus jamais.*

Démonstration. Ou moment où ces lignes sont exécutées, la valeur de *max* est ajoutée sur la pile *min* et ne pourra être retirée que par les lignes 34–35 lors de la suppression de la composante d'indice n' (ou inférieur).

Grâce au lemme 3, nous savons que $max \geq root_{n'}$ tant que la CFC n' n'a pas été retirée (ce qui n'arrive qu'une fois que $succ_{n'}$ et $fsucc_{n'}$ sont vidés, donc ne nous concerne pas). Ainsi, peu importe le nombre de fois où la ligne 13 sera appelée par la suite, nous avons $min_p \geq root_{n'}$ tant que la CFC n' n'a pas été retirée.

Le seul moyen d'ajouter des éléments à $succ_{n'}$ et $fsucc_{n'}$ serait d'appeler la fonction `merge` avec $t < root_{n'}$, mais la ligne 20 empêche cela. □

Notons $\varphi(x)$ l'indice de la CFC visitée contenant l'état numéroté x.

$$\varphi(x) = \max\{i \mid root_i \leq x\}$$

Lemme 5. *La fonction g définie par*

$$g : [\![0, p]\!] \to [\![0, n]\!]$$
$$i \mapsto \varphi(min_i)$$

est injective. C'est-à-dire que deux états numérotés min_{i_1} et min_{i_2} (avec $i_1 \neq i_2$) ne peuvent pas appartenir à la même CFC.

D'autre part, si $n > min_p$, $root_{\varphi(min_p)+1} = min_p + 1$, c'est-à-dire que min_p est le numéro du dernier état de la CFC dont la racine porte le numéro $root_{\varphi(min_p)}$.

Enfin, $root_{\varphi(min_p)} \leq min_p \leq max$.

Démonstration. Une fois que les lignes 12–13 ont été exécutées pour une CFC d'indice $n = n'$ donné, le lemme 4 page précédente assure que $succ_{n'}$ et $fsucc_{n'}$ ne peuvent plus croître. Comme $fsucc$ est maintenant vide et qu'il ne peut se remplir, la condition de la ligne 11 ne peut plus être vérifiée. La ligne 13 ne peut donc être exécutée qu'une fois par entrée de SCC, ce qui assure l'injectivité.

La seconde partie du lemme découle directement des lignes 27–29 et 32–33.

La dernière partie est facilement vérifiée :
- Lorsqu'un nouveau min_p est empilé ligne 13, on sait d'après le lemme 3 que $root_n \leq max$ donc $root_{\varphi(min_p)} = root_n < max = min_p$.
- Lorsque max est diminué ligne 33, on teste immédiatement lignes 34–35 s'il est nécessaire de supprimer le sommet de min. (L'injectivité de g nous assure qu'il n'est pas nécessaire de supprimer plus que le sommet.)

□

Comme dans la version sur les TGBA de l'algorithme, nous pouvons encore partionner les états en trois ensembles:
- Les états *actifs* sont ceux qui apparaissent dans H et y sont associés à une valeur non nulle,
- les états *retirés* sont ceux qui apparaissent dans H avec une valeur nulle,
- enfin, les états *non explorés* sont ceux qui n'apparaissent pas dans H.

Cependant dans cette version de l'algorithme un état *actif* peut redevenir *non exploré* lignes 41–42.

Les invariants suivants sont conservés par toutes les lignes de la fonction principale (lignes 9–23).

Proposition 20. *Pour tout entier $i \leq n$ le sous-graphe induit par les états de $\mathfrak{S}_i$ est une CFC. De plus, il existe un circuit dans cette CFC qui visite toutes les conditions d'acceptation de acc_i. Enfin, $\mathfrak{S}_0, \mathfrak{S}_1, \ldots, \mathfrak{S}_n$ est une partition de l'ensemble des états actifs.*

Proposition 21. $\forall i < n$, $\exists s \in \mathfrak{S}_i$, $\exists s' \in \mathfrak{S}_{i+1}$, $\exists p \in 2^{2^{AP}}$, $\langle s, p, la_{i+1}, s' \rangle \in \delta$.

Autrement dit, il existe une transition étiquetée par la_{i+1} entre les CFC d'indice i et $i+1$.

Proposition 22. *Il existe exactement max états actifs. Aucun état de H n'est associé à une valeur supérieure à max. Si deux états (différents) portent la même valeur dans H, cette valeur est 0.*

Cela signifie en particulier que pour toute valeur v comprise entre 1 et max il existe un unique état actif s tel que $H[s] = v$.

Proposition 23. *Pour tout entier $i \leq n$, l'ensemble rem_i contient tous les états de $\mathfrak{S}_i \setminus \{state_i\}$.*

Proposition 24. *Pour tout état* retiré *q,* $\mathrm{Acc}(A[\{q\}]) = \emptyset$.

Proposition 25. *Il n'existe pas d'état accessible depuis $state_n$ autour duquel il existerait un circuit acceptant traversant une condition d'acceptation de $\overline{acc_r}$.*

Proposition 26. *Toutes les transitions allant de $\mathfrak{S}_{\varphi(min_p)}$ à $\mathfrak{S}_{\varphi(min_p)+1}$ sont étiquetées par une condition d'acceptation de $\overline{acc_r}$. (Donc en particulier $la_{\varphi(min_p)+1} \cap \overline{acc_r} \neq \emptyset$.)*

Proposition 27. *$\forall j \geq \varphi(min_p)$, $acc_j \cap \overline{acc_r} = \emptyset$ et $\forall j > \varphi(min_p) + 1$, $la_j \cap \overline{acc_r} = \emptyset$, c'est-à-dire que les CFC construites après la dernière barrière et les transitions les reliant ne sont pas étiquetées par des conditions d'acceptation de $\overline{acc_r}$, exception faite de la première transition franchie après la dernière barrière (étiquetée par $la_{\varphi(min_p)+1}$).*

Quatre de ces invariants (20, 21, 23 et 24) correspondent aux propositions 13 à 16 page 124 de la preuve de l'*emptiness check* pour TGBA, à quelques détails près: parler du chemin de recherche n'a plus vraiment de sens dans la proposition 23 et la proposition 24 ne diffère de la proposition 16 page 124 que dans le fait que l'ensemble des états retirés ne forme pas nécessairement une union de CFC maximales.

Les deux premières propositions nous assurent que, si l'algorithme trouve un i tel que $\forall (l,u) \in \mathcal{F}$, $acc_i \in l \implies acc_i \in u$, alors $SCC[i]$ est une composante fortement connexe acceptante (prop. 20) et accessible (prop. 20 et 21). La proposition 24 justifie qu'aucun chemin acceptant n'existe si tous les états ont été *retirés*.

Démonstration. Les huit propositions (20–27) sont vérifiées quand la ligne 9 est atteinte pour la première fois. Comme $n = m = p = 0$, les propriétés 21, 23 et 26 sont trivialement vérifiées. $\mathfrak{S}_0$ contient seulement un état, donc la propriété 20 est vraie ; seul cet état est actif et $max = 1$ donc la propriété 22 est vraie. Aucun état n'a encore été *retiré*, donc la propriété 24 est vérifiée. $\overline{acc_r} = \emptyset$ donc la propriété 25 ne peut être fausse. Enfin SCC ne contient qu'une CFC triviale, avec $acc_0 = \emptyset$, donc la propriété 27 est satisfaite.

Quand une transition $\langle r, \_, a, d\rangle$ est retirée de $succ_m$, nous nous trouvons dans l'un des quatre cas suivants:

- $H[d] = 0$. Cela signifie que d est un état retiré. D'après la propriété 24, ni d ni aucun de ses descendants ne peut faire partie d'un chemin acceptant. Cet état est donc ignoré par l'algorithme. Comme aucune structure de donnée n'est altérée, toutes les propositions sont préservées.
- $d \notin H$ (ligne 18). d n'a jamais été exploré. Nous lui donnons donc un rang dans H et le considérons comme une CFC triviale en l'empilant sur SCC, et continuons le parcours en profondeur vers ses successeurs en l'empilant sur $todo$. Faire cela respecte les propriétés 20–22 sans affecter les propositions 23–25.
  Pour les deux dernières propriétés, nous distinguons deux cas, selon que le successeur vient réellement de $succ$ ou qu'il vient en fait de $fsucc$ après permutation ligne 12. Le lemme 2 page 175 nous assure que les successeurs de $succ$ ne sont pas étiquetés par des conditions d'acceptation de $\overline{acc_r}$, alors que ceux de $fsucc$ le sont.
  - Si le successeur vient vraiment de $succ$, on a $a \cap \overline{acc_r} = \emptyset$, et nous savons que les lignes 12–13 n'ont pas encore été franchies pour cette CFC. D'après le lemme 5

page 176, nous en déduisons que $min_p < root_{\varphi(min_p)+1} \leq root_n$. Cela permet donc d'assurer la propriété 27. La véracité de la proposition 26 n'est pas changée, car la ligne 19 ajoute une nouvelle composante après la $(\varphi(min_p)+1)^{\text{e}}$.

- Si le successeur vient de $fsucc$, on a $a \cap \overline{acc_r} \neq \emptyset$. Tous les successeurs de $succ$ ont été visités, donc la nouvelle CFC créée à la ligne 19 ne pourra être accessible depuis $\mathfrak{S}_{\varphi(min_p)}$ que par des transitions initialement dans $fsucc$, c'est-à-dire des transitions étiquetées par des conditions de $\overline{acc_r}$. Ceci assure la propriété 26.
  La première partie de la proposition 27 est vérifiée sans problème puisque la nouvelle valeur empilée pour acc_n est $\emptyset$. La seconde partie n'est pas affectée car la valeur de $la_{\varphi(p)+1)}$ n'y intervient pas.

– $H[d] > min_p$ est un état *actif*. Notons $root_i$ la plus grande racine de CFC telle que $root_i < H[d]$ et notons r_i l'état tel que $H[r_i] = root_i$ (l'existence de cet état est assurée par la proposition 22).
  D'après la proposition 20, r_i et d sont dans la même CFC. Comme r_i et s sont sur le chemin de recherche, r_i peut atteindre s. Comme nous sommes en train de considérer une transition entre s et d, nous en déduisons que r_i, s et d appartiennent à la même CFC.
  Nous pouvons donc fusionner toutes les CFC au dessus de $root_i$. La nouvelle CFC est l'union de $\mathfrak{S}_i, \mathfrak{S}_{i+1} \ldots, \mathfrak{S}_n$, et dans cette CFC il existe un circuit qui traverse les conditions d'acceptation $acc_i, acc_{i+1}, \ldots, acc_n$ de chacune des CFC fusionnées aussi bien que les conditions d'acceptation $la_{i+1}, la_{i+2}, \ldots, la_n$ qui les séparaient, ainsi que les conditions d'acceptation a sur la transition que l'on considère. (La figure 5.10 page 123 montre la situation avant la fusion.)
  La fonction `merge` s'occupe de fusionner ces conditions d'acceptation afin de préserver la proposition 20. Elle fusionne aussi les états de *rem* pour satisfaire la proposition 23. Les six autres propriétés ne sont pas affectées par cette opération.

– $H[d] \leq min_p$ le test de la ligne 20 échoue et aucune des structure de donnée n'est modifiée. L'ensemble des propositions reste donc vérifié. Ce cas correspond à une tentative de franchissement de ce que nous avons appelé une « barrière » dans notre introduction.

Après chaque appel de `merge`, c'est-à-dire chaque fois que l'on a créé une CFC non-triviale, l'algorithme vérifie si cette CFC valide les conditions d'acceptation (ligne 23). Dans ce cas, l'algorithme peut répondre immédiatement par la négative: l'automate n'est pas vide car il existe un circuit acceptant dans cette CFC qui est accessible depuis l'état initial.

Nous nous tournons maintenant vers le cas $succ_n \cup fsucc_n = \emptyset$, testé lignes 10–11 et conduisant ligne 15. Les propriétés du parcours en profondeur impliquent alors que tous les successeurs de $state_n$ ont été explorés, ainsi que leurs descendants. En d'autres termes la CFC $\mathfrak{S}_m$ est terminale. Au terme de la fonction `DFSpop`, les états ne seront plus actifs soit à cause des lignes 41–42, soit à cause des lignes 46–47. Il est donc nécessaire d'ajuster la valeur de max, ligne 33, pour satisfaire la propriété 22.

Nous envisageons maintenant les trois cas décrits page 168.

Le cas où la composante est non triviale et acceptante ne peut se produire: une telle composante aurait été détectée ligne 23 immédiatement après sa formation.

Dans le cas où $\exists i, l_i \in acc \wedge u_i \notin acc$, notons $bad = \{l_i \mid l_i \in acc \wedge u_i \notin acc\}$. Comme tous les descendants accessibles sans franchir de transition étiquetée par une condition de $\overline{acc_r}$ ont été visités par le DFS, nous pouvons affirmer qu'il n'existe pas dans les descendants des états de cette CFC de circuit acceptant traversant une transitions étiquetée par une condition d'acceptation de $bad \cup \overline{acc_r}$. Empiler cette union de conditions sur la pile *avoid* (ligne 43) n'invalidera pas la proposition 25.

Dans le cas où la composante est triviale, cela signifie qu'il n'existe pas de cycle autour de l'unique état de cette composante qui soit acceptant et qui ne traverse pas de conditions de $\overline{acc_r}$. Comme d'autre part la proposition 25 nous assure qu'il n'existe pas non plus de cycle acceptant traversant les conditions de $\overline{acc_r}$, nous en déduisons qu'aucun chemin acceptant ne peut passer par cet état. Il est donc correct de *retirer* l'état de la composante. lignes 46–47: la propriété 24 est forcément vérifiée.

Si l'algorithme termine ligne 24, la pile de recherche *SCC* est vide. D'après la propriété 20 cela signifie qu'il n'existe plus d'états actifs. Comme le DFS a exploré tous les états accessibles de l'automate, nous en concluons que tous les états accessibles ont été retirés. Finalement, la propriété 24 implique que $\mathrm{Acc}(A) = \emptyset$.

Nous avons montré que si l'algorithme termine ligne 23, alors il existe un chemin acceptant, et que s'il termine ligne 24 il n'y en a pas. La terminaison de l'algorithme est assurée par le fait qu'il s'agit d'un parcours en profondeur de l'automate.

En conséquence, cet algorithme retourne $\bot$ si et seulement si l'automate contient un chemin acceptant. □

7.7 Conclusion

Dans cette section nous avons introduit un nouvel algorithme pour l'*emptiness check* d'automates de Streett. Cet algorithme est une généralisation de celui présenté section 5.4.1 page 120, et il peut visiter chaque état jusqu'à r fois de plus que l'algorithme original (si r est le nombre de paires de conditions d'acceptation).

Comparons les deux approches que nous avons évoquées pour la vérification d'hypothèses d'équité forte:

Via LTL Les n hypothèses d'équité forte sont exprimées sous la forme d'une formule LTL. Cette formule est traduite en un TGBA de taille 3^n synchronisé avec le système et la propriété à vérifier. Le TGBA inspecté par l'algorithme d'*emptiness check* est jusqu'à 3^n fois plus gros.

Via Streett Les n hypothèses d'équité forte sont traduites sous la forme de conditions d'acceptation de Streett intégrées à l'espace d'état lors de sa génération. Cet espace d'état est synchronisé avec l'automate de la propriété à vérifier (interprété comme un automate de Streett) avant d'être soumis à l'*emptiness check* pour automates de Streett. Ce dernier peut être jusqu'à $n + 1$ fois plus lent que la version pour TGBA.

En changeant de type d'automate, nous avons donc échangé un surcoût exponentiel pour un surcoût linéaire.

Cet algorithme, comme tous les autres, a été implémenté dans la bibliothèque Spot (annexe A page 201).

Chapitre 8

Conclusion générale

Dans cette thèse nous avons abordé le *model checking* de formule de logique temporelle à temps linéaire en utilisant des automates de Büchi généralisés étiquetés sur les transitions (TGBA), ou des automates de Streett étiquetés sur les transitions. Ces automates sont pour l'instant assez peu utilisés: on leur préfère en général des automates de Büchi non généralisés dans lesquels les conditions d'acceptation portent sur les états, parce qu'il existe plus d'algorithmes les manipulant. Nous avons montré qu'il était tout à fait raisonnable (voir plus intéressant) d'utiliser des automates généralisés et basés sur les transitions d'un bout à l'autre du processus de vérification par l'approche automate. Il nous semble regrettable que leur utilisation ne soit pas plus commune.

8.1 De l'intérêt des TGBA

Dans le chapitre 3 nous avons présenté les TGBA et expliqué que, s'ils étaient aussi expressifs que les automates de Büchi, ils étaient plus concis. Selon nous leur intérêt se résume en trois points.

Étiquetés sur les transitions. Dans le chapitre 4 nous avons vu que la traduction d'un automate par la méthode des tableaux permettait de construire des automates de Büchi étiquetés au choix sur les états ou les transitions, mais que la version sur les transitions était plus compacte par construction. L'intérêt d'un étiquetage sur les transitions lors de la traduction a au reste été mentionnée par plusieurs auteurs au cours des 20 dernières années [96, 34, 58, 63, 123], même s'il ne s'agissait que d'une étape intermédiaire pour plusieurs d'entre-eux.

Conditions d'acceptation généralisées. Dans le chapitre 5 nous avons montré l'avantage de travailler avec des automates généralisés du point de vue de l'*emptiness check*. L'inconvénient de l'approche traditionnelle, qui consiste à dégénéraliser un automate avant de tester sa vacuité avec deux parcours en profondeur imbriqués, est qu'elle augmente la taille de cet automate de façon linéaire en dupliquant chaque état autant de fois qu'il existe de conditions d'acceptation. Les algorithmes d'*emptiness check* basés sur

les CFC qui sont insensibles au nombre de conditions d'acceptation et ne subissent donc pas ce surcoût. Même l'algorithme d'*emptiness check* par NDFS de Tauriainen [122], qui reste sensible au nombre de conditions d'acceptation, consomme moins de mémoire que l'approche par dégénéralisation.

Équité. Enfin nous avons vu chapitre 7 que les TGBA (resp. automates de Streett) permettent de représenter des hypothèses d'équité faible (resp. équité forte) sans surcoût sur la taille de l'automate à tester. Ceci est dû à la fois à l'étiquetage sur les transitions et à l'emploi de conditions d'acceptation généralisées. La prise en compte d'hypothèses d'équité forte sur le modèle, si elle n'a pas de surcoût sur la taille de l'automate, entraîne tout de même un surcoût sur la vérification (l'algorithme d'*emptiness check* est plus lent), mais celui-ci est moindre que le surcoût lié à l'expression des hypothèses d'équité en LTL.

8.2 Optimisations et nouveaux algorithmes

Outre le passage en revue des algorithmes de *model checking* du point de vue des TGBA, les contributions de cette thèse sont principalement des variantes ou améliorations d'algorithmes de traduction de formule LTL et d'*emptiness check* existants.

Chapitre 4 nous avons présenté de nouvelles règles de simplification de formules LTL inspirées de celle de Tauriainen [123]. Nous avons aussi implémenté plusieurs optimisations dans l'algorithme de traduction de Couvreur [34].

Chapitre 5 nous avons proposé l'optimisation des *poids*, qui s'applique à tout algorithme d'*emptiness check* basé sur des NDFS, et généralise une idée de Schwoon et Esparza [107]. Nous avons aussi évalué deux heuristiques visant à diriger l'*emptiness check* de Couvreur [34]. Enfin nous avons proposé des techniques pour le calcul de contre-exemples dans les algorithmes d'*emptiness check* généralisés.

Chapitres 6 et 7 nous avons introduit deux variantes d'*emptiness check* pour des approches légèrement différentes du *model checking* par automates. Ces algorithmes, bien que dérivés de celui de Couvreur peuvent être considérés comme nouveaux au sens où ils résolvent des problèmes différents. L'un utilise des tests d'inclusion afin de diriger la construction à la volée de l'espace d'état avec des techniques de réduction telles que les *symétries partielles* [67]. L'autre est une généralisation de l'*emptiness check* aux automates de Streett, qui permettent de prendre en compte des hypothèses d'équité forte.

8.3 Perspective à court terme: Ensembles persistants équitables

Les techniques basées sur les ensembles persistants tentent de réduire l'espace d'état en s'appuyant sur le fait que plusieurs entrelacements sont équivalents vis-à-vis de la propriété que l'on cherche à vérifier. Revenons sur l'exemple de client/serveur que nous avions utilisé tout au long du chapitre 2 et sur lequel nous voulions vérifier la formule

$$\mathsf{G}(d_1 \rightarrow \mathsf{F}\, r_1)$$

Sur cet exemple, l'exécution suivante:
– C_1 envoie une requête au serveur
– C_2 envoie une requête au serveur
– reste de l'exécution...
satisfera la formule de la même façon que l'exécution
– C_2 envoie une requête au serveur
– C_1 envoie une requête au serveur
– suite identique à la première exécution...
En d'autres termes, vis-à-vis de la formule que nous cherchons à vérifier, les règles de synchronisation (1) et (2) de la figure 2.2 page 18 peuvent être permutées dans toute exécution du système. Nous pouvons tirer parti de telles équivalences lors de la génération de l'espace d'état pour réduire les choix possibles.

Par exemple, la figure 8.1 page suivante montre une structure de Kripke équivalente à celle de la figure 2.4 page 21 du point de vue de la validité de la formule $\mathsf{G}(d_1 \rightarrow \mathsf{F}\, r_1)$. Douze transitions superflues ont pu être retirées et cela a rendu deux états inaccessibles (ils ont été supprimés de la figure).

Plusieurs approches similaires ont été développées pour réduire d'espace d'état du système de cette façon: les *stubborn sets* [132, 142], les *persistent sets* [64], les *ample sets* [101] ou encore les *graphes de pas couvrants* [143]. L'idée est toujours de tirer parti des permutations possibles pour réduire l'ensemble des successeurs qui doivent être explorés après un état.

Par exemple, dans l'état initial du système de la 2.4 page 21, les règles (1) et (2) de la figure 2.2 page 18 peuvent toutes deux être appliquées. Si nous appliquons (2), la règle (1) restera applicable. Comme l'ordre de franchissement des transitions correspondant à ces deux règles n'a pas d'influence sur la propriété à vérifier, nous pouvons décider de n'appliquer que (2) dans cet état. C'est ce qui est fait figure 8.1 page suivante.

Selon ces critères, il serait possible d'ignorer toujours la règle (1) et de ne faire évoluer que le client C_2 et le serveur S. Ce problème, connu sous le nom de *ignoring problem*, est corrigé par l'ajout d'une nouvelle contrainte appelée *proviso*. Un proviso classique consiste à explorer systématiquement toutes les transitions sortantes d'un état atteint lorsque l'on ferme un circuit, de cette façon une règle de synchronisation ne peut pas être toujours ignorée.

Dans notre implementation des *stubborn sets*, l'*emptiness check* (basé sur celui présenté section 5.4.1 page 120) retient l'ensemble de toutes les règles de synchronisation qui ont été ignorées dans une composante fortement connexe. Lorsque plusieurs composantes sont fusionnées, ces ensembles sont intersectés. Si l'ensemble n'est pas vide au moment où une CFC doit être dépilée, l'algorithme construit un nouveau *stubborn set* à partir des règles ignorées. L'exemple de la figure 8.1 a été généré par cette implémentation.

Nous aimerions prendre en compte des hypothèses d'équité sur le modèle lors de la génération des stubborns set. Pour l'instant nous disposons d'une implémentation permettant de faire le *model checking* d'un système de transitions étiquetées *soit* en utilisant les *stubborn sets* discutés ci-dessus, *soit* en faisant des hypothèses d'équité comme celle présentées section 7.5 page 167, mais pas les deux. La difficulté vient du fait que l'emploi des conditions d'acceptation dont nous avons discuté section 7.5 n'étiquette pas toutes

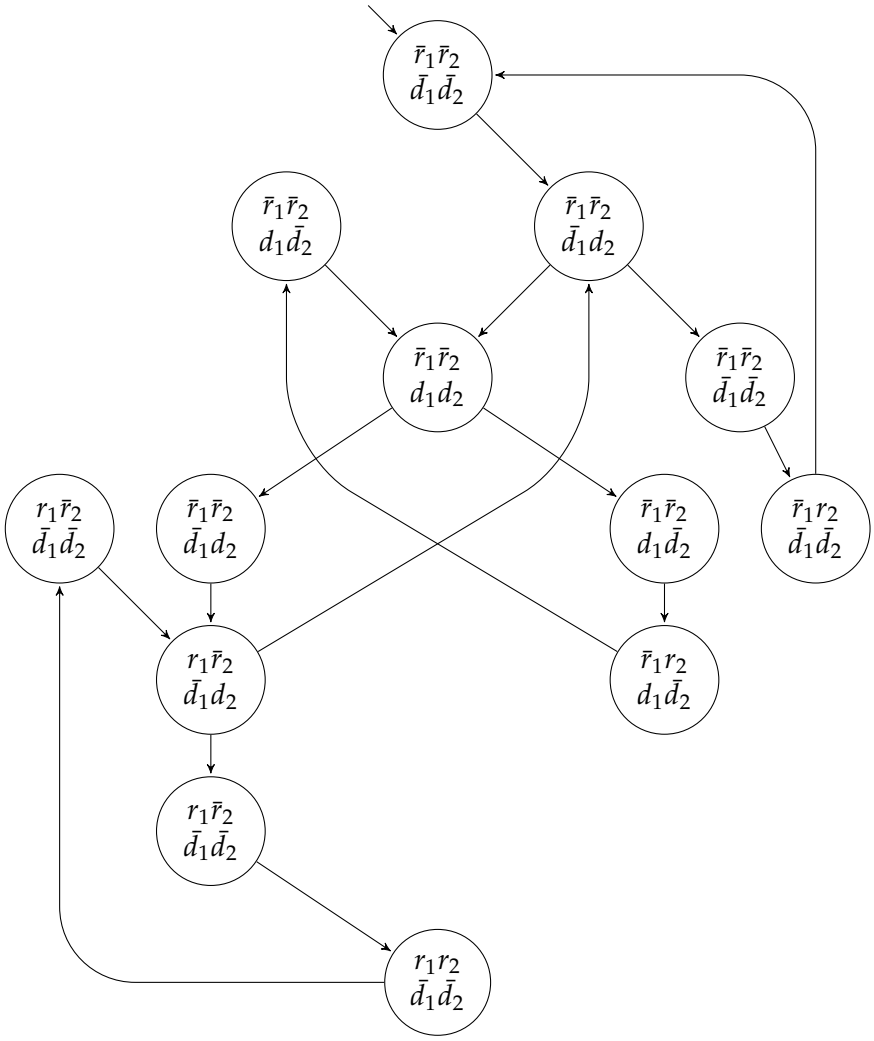


FIG. 8.1: Structure de Kripke réduite. À comparer à la figure 2.4 page 21.

les instances d'une règle de synchronisation par les mêmes conditions d'acceptation. Il faudrait donc améliorer le proviso pour ne pas considérer uniquement un ensemble de règles ignorées par une CFC, mais plutôt un ensemble de paires «(règle, conditions d'acceptation)».

8.4 Vers une parallélisation de l'*emptiness check*

Les algorithmes d'*emptiness check* utilisés dans l'approche automate pour le *model checking* de formules LTL sont basés sur des parcours en profondeur. À la différence du parcours en largeur d'abord, le parcours en profondeur d'abord se parallélise difficilement.

Une approche de la parallélisation de l'*emptiness check* consiste à distribuer l'espace d'état qui doit être exploré. Barnat [12], chapitre 4 donne un état de l'art des techniques de *model checking* LTL basées sur de la mémoire distribuée.

Une autre idée consiste à envisager comment les algorithmes d'*emptiness check* pourraient tirer parti de plusieurs processeurs qui partagent leur mémoire. Ce type d'architecture est de plus en plus commun avec la généralisation des machines multi-processeurs et des processeurs multi-core. Holzmann et Bošnački [76] ont récemment publié les premiers résultats d'une adaptation du NDFS utilisé par le *model checker* Spin à ce type d'architecture. Le premier parcours en profondeur est effectué sur un processeur ; lorsqu'il doit lancer un second parcours en profondeur pour chercher un cycle, cette tâche est lancée sur un autre processeur et le parcours principal continue sa recherche. Ce type de parallélisation pourrait facilement être adapté à l'algorithme Tau03 opt de la section 5.3.3 page 115).

De façon moins évidente, une technique similaire pourrait être employée pour paralléliser l'*emptiness check* d'automate de Streett présenté dans la section 7.6 page 168. Lorsqu'une CFC est retirée de la pile et que l'algorithme décide qu'elle doit être reparcourue en évitant certaines conditions d'acceptation, ce nouveau parcours peut être effectué par un processeur différent.

Nous avons travaillé sur un type de parallélisation différent, basé sur une idée de Denis Poitrenaud[1]. Le principe est que plusieurs instances d'un algorithme d'*emptiness check* vont explorer l'automate en parallèle, mais avec des ordres de parcours différents. Tous les *emptiness checks* sont basés sur un parcours en profondeur, dans lequel le choix de l'ordre des successeurs n'a pas d'importance: nous pouvons donc ordonner ces successeurs de façon aléatoire en espérant que les *emptiness checks* explorent ainsi différentes parties de l'automate. Les différentes instances peuvent alors s'envoyer des informations utiles telles que « telle CFC n'est pas acceptante » ou « il n'existe pas de cycle acceptant à partir de cet état » qui pourront abréger les parcours des autres instances . Nous avons implémenté cette technique avec Cou99 (section 5.4.1 page 120) et CVWY90. Des difficultés techniques (la bibliothèque de BDD utilisée par Spot n'est pas réentrante) nous ont forcés à utiliser des processus distincts qui communiquent par tubes au lieu de *threads* qui communiquent par mémoire partagée. Bien que les résultats ne soient pour l'instant

[1] Idée qu'il a eue suite à une discussion sur la technique de résolution « parallèle » de problèmes de Sudoku appliquée par les secrétaires d'un cabinet médical de Papeete.

pas très probants, nous avons pu construire quelques cas où la communication entre les différents *emptiness checks* conduit à une réponse plus rapide.

8.5 Problème ouvert: traduction efficace de LTL vers Streett

Dans le chapitre 7 page 159 nous avons montré comment un modèle pouvait être vérifié sous des hypothèses d'équité forte en le représentant par un automate de Streett. Cet automate était alors synchronisé avec l'automate de Büchi (vu comme un cas particulier d'automate de Streett) représentant la négation de la formule à vérifier, pour être ensuite exploré par un algorithme d'*emptiness check* dédié.

Cette approche n'est pas optimale puisque nous avons aussi montré que certaines formules LTL pouvaient être représentées de façon plus compacte. Par exemple, l'automate de Streett de la figure 7.4 page 165 est plus intéressant que le TGBA de la figure 7.3 page 163 car il est à la fois déterministe et plus concis. Comme dans ce contexte l'agorithme d'*emptiness check* manipule des automates de Streett il semble naturel de se demander s'il ne serait pas possible de traduire la formule à vérifier dans ce formalisme.

À notre connaissance, personne n'a encore proposé d'algorithme de traduction directe entre une formule LTL et un automate de Streett. Il est possible de transformer un automate Büchi en un automate de Streett déterministe [105] mais d'une part cette opération complexe ne réduit pas la taille de l'automate, d'autre part elle supposerait que la formule ait été préalablement traduite en automate de Büchi.

Bien que cela ne soit pas satisfaisant, il serait toujours possible d'extraire certaines sous-formules que l'on sait traduire en automates de Streett. Par exemple si la formule LTL à traduire est de la forme

$$\varphi \wedge \left(\bigwedge_{i=1}^{n} \mathsf{GF}\, a_i \rightarrow \mathsf{GF}\, b_i \right)$$

nous pouvons la traduire comme un produit d'automates $A_\varphi \otimes A_{\bigwedge_{i=1}^{n} \mathsf{GF}\, a_i \rightarrow \mathsf{GF}\, b_i}$ où A_φ a été construit avec un algorithme de traduction de formule LTL en automate de Büchi et $A_{\bigwedge_{i=1}^{n} \mathsf{GF}\, a_i \rightarrow \mathsf{GF}\, b_i}$ serait un automate de Streett comparable à celui de la figure 7.3 page 163.

Bibliographie

[1] Jean-Raymond Abrial. *The B Book*. Cambridge University Press, October 1996. 1.2

[2] K. Ajami, S. Haddad, et J-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998. 2.7

[3] Bowen Alpern et Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. 2.3

[4] Bowen Alpern et Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987. 2.3

[5] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture notes, October 1997. 4.2

[6] André Arnold. *Finite transition systems. Semantics of communicating systems*. Prentice-Hall, 1994. 2.1

[7] Soheib Baarir, Serge Haddad, et Jean-Michel Ilié. Exploiting partial symmetries in well-formed nets for the reachability and the linear time model checking problems. In *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, pages 223–228, Reims, France, September 2004. 6.5, 6.5.2, 6.5.3, A.2.1

[8] Souheib Baarir. *Exploitation des symétries partielles pour la vérification et l'évaluation de performances des systémes finis*. PhD thesis, Université Pierre et Marie Curie (Paris 6), France, May 2007. 6.5.1

[9] Souheib Baarir et Alexandre Duret-Lutz. Emptiness check of powerset Büchi automata. Technical report 2006/003, Université Pierre et Marie Curie, LIP6-CNRS, Paris, France, October 2006. 6, 6.3.2, 6.5.1

[10] Souheib Baarir et Alexandre Duret-Lutz. Emptiness check of powerset Büchi automata. In *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD'07)*, July 2007. To appear. 6

[11] Souheib Baarir et Alexandre Duret-Lutz. Test de vacuité pour automates de Büchi ensemblistes avec tests d'inclusion. In *Actes du 6e Colloque Francophone sur la Modélisation des Systèmes Réactifs (MSR'07)*, October 2007. À paraître. 6

[12] Jiří Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University Brno, 2005. 8.4

[13] Howard Barringer, Michael D. Fisher, et Graham D. Gough. Fair SMG and linear time model checking. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1989. 4.1

[14] Patrick Behm, Paul Benoit, Alain Faivre, et Jean-Marc Meynadier. Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, et Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387, Toulouse, France, September 1999. Springer-Verlag. 1.2

[15] Mordechai Ben-Ari, Zohar Manna, et Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 164–176. ACM, 1981. 4.1.3

[16] Evert Willem Beth. *La crise de la raison et la logique*. Gauthier-Villars, 1957. 3, 4.1.3

[17] Evert Willem Beth. *The Foundations of Mathematics*. North Holland, 1959. Second edition revised in 1965. 4.1, 3, 4.1.3

[18] Roderick Bloem. *Search Techniques and Automata for Symbolic Model Checking*. PhD thesis, University of Colorado, 2001. 4.3.3, 4.3.3, 4.4

[19] Ahmed Bouajjani, Jean-Claude Fernandez, Nicolas Halbwachs, Pascal Raymond, et Christophe Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992. 6.1

[20] Fabrice Bouquet, Laurent Henocque, et Philippe Jégou. Énumération et représentation d'impliquants premiers. In *Actes des cinquièmes Journées Nationales sur la résolution pratique de Problèmes NP-Complets (JNPC'99)*, pages 179–188, Lyon, France, June 1999. 4.2

[21] Daniel Brand et Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. 2.1

[22] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. 4.2

[23] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Berkley, 1960*, pages 1–11. Standford University Press, 1962. Republished in [87]. 3.1.3, 3.3, 3.3.2, 3.3.6, 3.3.6

[24] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, et L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press. 4.1

[25] Ivana Černá et Radek Pelánek. Relating hierarchy of temporal properties to model checking. In Branislav Rovan et Peter Vojtáă, editors, *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'03)*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327, Bratislava, Slovak Republic, August 2003. Springer-Verlag. 5.2, 1

[26] Edward Y. Chang, Zohar Manna, et Amir Pnueli. Characterization of temporal property classes. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, pages 474–486, London, UK, 1992. Springer-Verlag. 1

[27] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, et Serge Haddad. On well-formed coloured nets and their symbolic reachability graph. In K. Jensen et G. Rozenberg, editors, *Procedings of the 11th International Conference on Application and Theory of Petri Nets. Paris, France, June 1990. Reprinted in High-Level Petri Nets. Theory and Application.* Springer-Verlag, 1991. 6.5

[28] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, et Serge Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1–2):39–65, April 1997. 6.5, A.2.1

[29] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, et A. Prasad Sistla. Efficient decision procedures for model checking of linear time logic properties. In *Proceedings of the Tenth Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, 1998. 2.7

[30] Edmund M. Clarke, Orna Grumberg, et Doron A. Peled. *Model Checking*. The MIT Press, 2000. 1.2, 3.3.3

[31] Edmund M. Clarke et Jeannette M. Wing. Formal methods: State of the art and future. *ACM Computing Surveys*, 28(4):626–643, 1996. 1.2

[32] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, et Mihalis Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In Edmund M. Clarke et Robert P. Kurshan, editors, *Proceedings of the 2nd international workshop on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer-Verlag, 1991. 5.2, 5.2, 5.3, 5.5, 5.7

[33] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, et Mihalis Yannakakis. Memory-efficient algorithm for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992. 5.2, 5.2, 5.7

[34] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In Jeannette M. Wing, Jim Woodcock, et Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, September 1999. Springer-Verlag. Remerciements, 1.5, 3.3.4, 3.4, 4.1, 4.1, 4.1.3, 4.1.3, 4.1.4, 4.2, 4.13, 4.6, 5.2, 5.2, 5.2, 5.4, 5.4.1, 5.8, 5.4.1, 5.5, 7.6, 8.1, 8.2

[35] Jean-Michel Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In Pierre Leroux, editor, *Actes du Colloque LaCIM 2000*, volume 27 of *Publications du LaCIM*, pages 131–140, Montréal, August 2000. Université du Québec à Montréal. 4.1, 4.1, 4.1.1, 4.1.2, 4.4, 7.4

[36] Jean-Michel Couvreur. Contribution à l'algorithmique de la vérification. Habilitation à diriger des recherches, ENS Cachan, 2004. 4.1.3

[37] Jean-Michel Couvreur, Alexandre Duret-Lutz, et Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In Patrice Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, August 2005. 3.3.4, 5, 5.2, 5.2, 5.3.2

[38] Jean-Michel Couvreur, Sébastien Grivet, et Denis Poitrenaud. Designing a LTL model-checker based on unfolding graphs. In *Proceedings of the 21th International Conference on Applications and Theory of Petri Nets (ICATPN'00)*, volume 2075 of *Lecture Notes in Computer Science*, Aarhus, Denmark, June 2000. Springer-Verlag. 2.7, 6.1, 6.6

[39] Marco Daniele, Fausto Giunchiglia, et Moshe Y. Vardi. Improved automata generation for Linear Temporal Logic. In N. Halbwachs et D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999. 4.1, 4.1, 4.1.3

[40] René G. de Vries et Jan Tretmans. On-the-fly conformance testing using Spin. In *Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker (SPIN'98)*, ENST 98 S 002, pages 115–128, Paris, France, November 1998. Ecole Nationale Supérieure des Télécommunications. 1.1

[41] Michel Diaz. *Réseaux de Petri, Modèles fondamentaux*. Traité IC2, série Informatique et systèmes d'information. Hermes Science, June 2001. 1.2, 2.1, A.2.1

[42] Edsger Wybe Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF, May 1973. 5.2, 5.4

[43] Edsger Wybe Dijkstra. Finding the maximal strong components in a directed graph. In *A Discipline of Programming*, chapter 25, pages 192–200. Prentice-Hall, 1976. 5.2, 5.4

[44] Alexandre Duret-Lutz et Denis Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83, Volendam, The Netherlands, October 2004. IEEE Computer Society Press. 3.3.4, 4.1.3

[45] Matthew B. Dwyer, George S. Avrunin, et James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, March 1998. ACM Press. 4.3.3, 5.5

[46] Steven Eker, José Meseguer, et Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN'03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 230–2–34, Portland, Oregon, USA, May . Springer-Verlag. 3.2.2

[47] Javier Esparza. Verification of systems with an infinite state space. In F. Cassez, C. Jard, B. Rozoy, et M. Dermot, editors, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP'00)*, volume 2067 of *Lecture Notes in Computer Science*, pages 183–186. Springer-Verlag, 2001. 1.2

[48] Javier Esparza, Stefan Römer, et Walter Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002. 2.7

[49] Kousha Etessami et Gerard J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (Concur'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167, Pennsylvania, USA, 2000. Springer-Verlag. 4.1, 4.1, 4.3.3, 4.3.3, 4.4, 4.4, 5.5

[50] Kousha Etessami, Thomas Wilke, et Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In Fernando Orejas, Paul G. Spirakis, et Jan van Leeuwen, editors, *Proceedings of the 28th international colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707, Crete, Greece, July 2001. Springer-Verlag. 4.1, 4.4

[51] Melving Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 2nd edition, 1996. 4.1.3

[52] Nissim Francez. *Fairness*. Springer-Verlag, 1986. 7.2, 7.2

[53] Ehud Friedgut, Orna Kupferman, et Moshe Y. Vardi. Büchi complementation made tighter. In *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'04)*, volume 3299 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2004. 3.3.6

[54] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra et Zhe Dang, editors, *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA'03)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48, Santa Barbara, California, July 2003. Springer-Verlag. 4.1, 4.1.4, 4.4

[55] Carsten Fritz. Concepts of automata construction from LTL. In G. Sutcliffe et A. Voronkov, editors, *Proceedings of the 12th Internation Conference on Logic for Programming, Artifical Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 728–742, Montego Bay, Jamaica, 2005. Springer-Verlag. 4.1.4

[56] Carsten Fritz et Thomas Wilke. State space reductions for alternating Büchi automata: Quotienting by simulation equivalences. In Manindra Agrawal et Anil Seth, editors, *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'2002)*, volume 2556 of *Lecture Notes in Computer Science*, pages 157–168, Kanpur, India, 2002. 4.1.4, 4.4

[57] Paul Gastin, Pierre Moro, et Marc Zeitoun. Minimization of counterexamples in SPIN. In S. Graf et L. Mounier, editors, *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108, April 2004. 5.2, 5.2, 5.3, 5.6.3

[58] Paul Gastin et Denis Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, et A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer-Verlag. 3.2.2, 3.4, 4.1, 4.1.4, 4.4, 4.4, 8.1

[59] Jaco Geldenhuys et Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In Kurt Jensen et Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, 2004. 5.2, 5.2, 5.4, 5.4.1, 5.4.2, 5.5, 5.5, 5.5

[60] Jaco Geldenhuys et Antti Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, November 2005. Conference paper selected for journal publication. 5.2, 5.2, 5.4, 5.4.1, 5.6

[61] Rob Gerth, Doron Peled, Moshe Y. Vardi, et Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95)*, pages 3–18, Warsaw, Poland, 1996. Chapman & Hall. 4.1, 4.1, 4.1.3, 4.1.3, 4.4, 8.5

[62] Dimitra Giannakopoulou et Flavio Lerda. Efficient translation of LTL formulæ into Büchi automata. Technical Report 01.29, Research Institute for Advanced Computer Science, June 2001. Later republished as [63]. 4.1

[63] Dimitra Giannakopoulou et Flavio Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In D.A. Peled et M.Y. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, Texas, November 2002. Springer-Verlag. 3.3.4, 3.4, 4.1, 8.1, 8.5

[64] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. 2.7, 8.3

[65] Patrice Godefroid et Gerard J. Holzmann. On the verification of temporal properties. In André A. S. Danthine, Guy Leduc, et Pierre Wolper, editors, *Proceedings of the 13th IFIP TC6/WG6.1 International Symposium on Protocol Specification, Testing,*

and Verification (PSTV'93), volume C-16 of *IFIP Transactions*, pages 109–124, Liege, Belgium, May 1993. North-Holland. 5.2, 5.2, 5.3, A.2

[66] Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, et Moshe Y. Vardi. On complementing nondeterminisitic Büchi automata. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, 2003. 3.3.6

[67] Serge Haddad, Jean-Michel Ilié, et Khalil Ajami. A model checking method for partially symmetric systems. In *Proceedings of the International Conference on Formal Description techniques: theory, application and tools (FORTE-PSTV'00)*, October 2000. 6.1, 6.1, 6.5, 6.5.1, 6.5.1, 8.2

[68] Serge Haddad, Jean-Michel Ilié, et Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In F. Wang, editor, *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'04)*, volume 3299 of *Lecture Notes in Computer Science*, pages 198–210, National Taiwan University, Taiwan, October 2004. Springer-Verlag. 6.1, 6.6

[69] Serge Haddad et François Vernadat. Vérification de propriétés spécifiques. In Michel Diaz, editor, *Vérification et mise en œuvre des réseaux de Petri*, Traité IC2, série Informatique et systèmes d'information, chapter 1. Hermes Science, January 2003. 4.1.3

[70] Moritz Hammer, Alexander Knapp, et Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs et Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, UK, April 2005. Springer-Verlag. 4.1.4, 5.2, 5.2, 5.4, 5.4.1, 5.5

[71] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software Practice and Experience*, 18(2):137–161, 1988. 5.7

[72] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on software Engineering*, 23(5):279–295, May 1997. 5.5

[73] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, November 1998. 5.7

[74] Gerard J. Holzmann. Software model checking. NATO Summer School, pages 309–355, Marktoberdorf, Germany, August 2000. IOS Press Computer and System Sciences. 1.2

[75] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. 2.1, 4.4, 5.5, 5.7, A.2.1

[76] Gerard J. Holzmann et Dragan Bošnački. Multi-core model checking with SPIN. In *1st Workshop on Tools, Operating Systems and Programming Models for Developing Reliable Systems (TOPMoDelS'07)*. 8.4

[77] Gerard J. Holzmann et Doron Peled. An improvement in formal verification. In *Proceeding of the 7th IFIP WG 6.1 International Conference on Formal Description Techniques (FORTE'94)*, volume 6 of *IFIP Conference Proceedings*, pages 109–124, Berne, Switzerland, 1994. Chapman & Hall. 5.2

[78] Gerard J. Holzmann, Doron A. Peled, et Mihalis Yannakakis. On nested depth first search. In Jean-Charles Grégoire, Gerard J. Holzmann, et Doron A. Peled, editors, *Proceedings of the 2nd Spin Workshop*, volume 32 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, May 1996. 5.2, 5.2, 5.3

[79] Jérôme Hugues, Yann Thierry-Mieg, Fabrice Kordon, Laurent Pautet, Soheib Barrir, et Thomas Vergnaud. On the formal verification of middleware behavioral properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume 133 of *Electronic Notes in Theoretical Computer Science*, pages 139–157. Elsevier Science Publishers, September 2004. 6.5.3

[80] Moshe Y. Vardi Joseph Y. Halpern. Model checking vs. theorem proving: A manifesto. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 325–334, Cambridge, MA, USA, April 1991. Morgan Kaufmann Publishers. 1.2

[81] Tommi Junttila. *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms*. PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2003. 2.7

[82] Ynoit Kesten et Amir Pnueli. A compositional approach to CTL* verification. *Theoretical Computer Science*, 331(2–3):397–428, February 2005. 4.1, 4.1.2

[83] Yonit Kesten, Zohar Manna, Hugh McGuire, et Amir Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubetis, editor, *Proceedings of the 5th Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 1993. 4.1

[84] Yonit Kesten, Amir Pnueli, et Moshe Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences*, 62(4): 668–690, 2001. 7.2

[85] Nils Klarlund. Progress measures for complementation of ω-automata with applications to temporal logic. In *Proceedings of the 32nd annual symposium on Foundations of computer science (FOCS'91)*, pages 358–367, San Juan, Puerto Rico, 1991. IEEE Computer Society Press. 3.3.6

[86] Stephen C. Kleene. *Mathematical Logic*. Wiley, New York, 1967. 6.2

[87] Sounders Mac Lane et Dirk Siefkes, editors. *The Collected Works of J. Richard Büchi*. Springer-Verlag, 1990. 8.5

[88] Timo Latvala. Model checking linear temporal logic properties of Petri nets with fairness constraints. Research Report A67, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, January 2001. 7.2

[89] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets (ICATPN'01)*, volume 2075 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 2001. 7.2

[90] Timo Latvala et Keijo Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):1–19, 2000. 5.6.1, 7.6

[91] Yige Li, Kanglin Xie, et Tao Hao. Combining Couvreur's algorithm with bitstate-hashing for emptiness check. In *Proceedings of the First International Multi-Symposium on Computer and Computational Sciences (IMSCCS'06)*, volume 2, pages 283–286, 2006. 5.2, 5.2, 5.4.1, 5.7, A.4

[92] Orna Lichtenstein et Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. ACM, 1985. 4.1, 5.2, 5.2, 5.4

[93] Oded Maler, Dejan Nickovic, et Amir Pnueli. From mitl to timed automata. In *Proceedings of the 4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer-Verlag, September 2006. 4.1.2

[94] Stephan Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, et M. Dermot, editors, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP'00)*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001. 1.2

[95] Stephan Merz et Ali Sezgin. Emptiness of linear weak alternating automata. Technical report, LORIA, December 2003. 5.4.1

[96] Max Michel. Algèbre de machines et logique temporelle. In Max Fontet et Kurt Mehlhorn, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS'84)*, volume 166 of *Lecture Notes in Computer Science*, pages 287–298, Paris, April 1984. 3.3.4, 3.4, 4.1, 4.1.2, 4.1.2, 4.1.2, 4.1.2, 5, 8.1

[97] Shin-ichi Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of the third Synthesis and Simulation and Meeting International Interchange workshop (SASIMI'92)*, pages 64–73, Kobe, Japan, April 1992. 4.2, 4.13

[98] Satoru Miyano et Takeshi Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984. 4.1, 6

[99] David E. Muller et Paul E. Shupp. Alternating automata on infinite objects : Determinacy and rabin's theorem. In Maurice Nivat et Dominique Perrin, editors, *Proceedings of the École de Printemps d'Informatique Theorique on Automata on Infinite Words*, volume 192 of *Lecture Notes in Computer Science*, pages 100–107. Springer, 1984. 4.1.4

[100] Denis Oddoux. *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires*. PhD thesis, Universitée Paris 7, Paris, France, December 2003. 3.3.3, 4.1.4

[101] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994. 2.7, 8.3

[102] Monika Rauch Henzinger et Jan Arne Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, volume 1097 of *Lecture Notes in Computer Science*, pages 16–27, Reykjavík, Iceland, July 1996. Springer-Verlag. 7.6

[103] Mauno Rönkkö. LBT: LTL to Büchi conversion. `http://www.tcs.hut.fi/Software/maria/tools/lbt/`, 1999. Implements [61]. 4.1.3, 4.4

[104] Shmuel Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, March 1989. 3.3.6, 7.3

[105] Shmuel Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proceedings of the 24th ACM Symposium on Theory of Computing*. ACM, May 1992. 3.3.6, 8.5

[106] Stefan Schwoon et Javier Esparza. A note on on-the-fly verification algorithms. Technical report, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, November 2004. 5.2, 5.2, 5.2, 5.3.3

[107] Stefan Schwoon et Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs et Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, April 2005. Springer-Verlag. 5.2, 5.2, 5.2, 5.3, 5.3.1, 5.2, 5.3.1, 5.3.2, 5.3.3, 5.4.1, 5.5, 5.5, 8.2

[108] Roberto Sebastiani et Stefano Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. Technical Report #DIT-03-041, University of Trento, July 2003. Extended version of [109]. 4.1, 4.3.1, 4.3.2, 4.4

[109] Roberto Sebastiani et Stefano Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In G. Goos, J. Hartmanis, et J. van Leeuwen, editors, *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *Lecture Notes in Computer Science*, pages 126–140, L'Aquila, Italy, October 2003. Springer-Verlag. 4.1, 4.3.1, 4.3.2, 4.4, 4.4, 8.5

[110] Roberto Sebastiani, Stefano Tonetta, et Moshe Y. Vardi. Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking. In Kousha

Etessami et Sriram K. Rajamani, editors, *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363, Edinburgh, Scotland, UK, July 2005. Springer-Verlag. 7.4, 7.4, A.4

[111] Raymon M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. 4.1, 3

[112] Fabio Somenzi et Roderick Bloem. Efficient Büchi automata for LTL formulæ. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263, Chicago, Illinois, USA, 2000. Springer-Verlag. 4.1, 4.1, 4.3.3, 4.3.3, 4.4, 5.5

[113] Ulrich Stern et David L. Dill. Improved probabilistic verification by hash compaction. In P.E. Camurati et H. Eveking, editors, *Proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'95)*, volume 987 of *Lecture Notes in Computer Science*, pages 206–224. Springer-Verlag, 1995. 5.7

[114] Deian Tabakov et Moshe Y. Vardi. Model checking büchi specifications. In *Proceedings of the 1st International Conference on Language and Automata Theory and Applications (LATA'07)*, Lecture Notes in Computer Science. Springer-Verlag, April 2007. To appear. 3.3.6

[115] Robert Tarjan. Depth-first search and linear graph algorithms. In *Conference records of the 12th Annual IEEE Symposium on Switching and Automata Theory*, pages 114–121. IEEE, October 1971. Later republished as [116]. 5.2, 5.4

[116] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. 5.2, 5.4, 8.5

[117] Heikki Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulæ into Büchi automata. In H-D. Burkhard, L. Czaja, H-S. Nguyen, et P. Starke, editors, *Proceedings of the Concurrency, Specification and Programming 1999 Workshop (CS&P'99)*, pages 251–262, Warsaw, Poland, September 1999. 4.4, 5.5

[118] Heikki Tauriainen. Automated testing of Büchi automata translators for Linear Temporal Logic. Research Report A66, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2000. Reprint of Master's thesis. 4.4

[119] Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. Research Report A79, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, July 2003. 5.2, 5.2, 5.3.3, 5.5

[120] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003. Reprint of Licentiate's thesis. 4.1, 5.2, 5.2, 5.3, 5.3.2, 5.3.3, 5.6, 5.5, 5.5

[121] Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 165–174. IEEE Computer Society, June 2004. 5.2, 5.2, 5.3, 5.3.3

[122] Heikki Tauriainen. A note on the worst-case memory requirements of generalized nested depth-first search. Research Report A96, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, September 2005. 5.2, 5.2, 5.3.3, 5.5, 8.1

[123] Heikki Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, Espoo, Finland, September 2006. 3.4, 4.1, 4.1.4, 4.3.3, 4.3.3, 5.2, 5.2, 5.3.3, 5.5, 5.5, 8.1, 8.2, A.4

[124] Heikki Tauriainen et Keijo Heljanko. Testing SPIN's LTL formula conversion into Büchi automata with randomly generated input. In K. Havelund, J. Penix, et W. Visser, editors, *Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN'2000)*, volume 1885 of *Lecture Notes in Computer Science*, pages 54–72, Stanford University, California, USA, August 2000. Springer-Verlag. 4.4

[125] Heikki Tauriainen et Keijo Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70, 2002. 4.4

[126] Yann Thierry-Mieg, Claude Dutheillet, et Isabelle Mounier. Automatic symmetry detection in well-formed nets. In Wil van der Aalst et Eike Best, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN'03)*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101, Eindhoven, The Netherlands, June 2003. Springer-Verlag. 2.7, A.2.1

[127] Yann Thierry-Mieg, Jean-Michel Ilié, et Denis Poitrenaud. A symbolic symbolic state space representation. In *Proceedings of the 24nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, Madrid, Spain, September 2004. 6.6

[128] Xavier Thirioux. Simple and efficient translation from LTL formulas to Büchi automata. In Rance Cleaveland et Hubert Garavel, editors, *Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66(2) of *Electronic Notes in Theoretical Computer Science*, Málaga, Spain, July 2002. Elsevier. 4.1, 4.2

[129] Wolfgang Thomas. Languages, automata, and logic. Technical Report 9607, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Germany, May 1996. 3.2

[130] Wolfgang Thomas. Complementation of Büchi automata revisited. In J. Karhumäki, H.A. Maurer, G. Paun, et G. Rozenberg, editors, *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–120. Springer-Verlag, 1999. 3.3.6

[131] Jan Tretmans. Testing concurrent systems: A formal approach. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999. 1.1

[132] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (ICATPN'91)*, volume 618 of *Lecture Notes in Computer Science*, pages 491–515, London, UK, 1991. Springer-Verlag. 2.7, 8.3

[133] Antti Valmari. On-the-fly verification with stubborn sets. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*, pages 397–408, London, UK, 1993. Springer-Verlag. 3.3.7

[134] Antti Valmari. The state explosion problem. In W. Reisig et G. Rozenberg, editors, *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998. 2.7, 6.1

[135] Moshe Y. Vardi. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press, 1986. 3.3

[136] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller et Graham M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop (Banff'94)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, Banff, Alberta, Canada, 1996. Springer-Verlag. 1.4, 2.5, 3.3.6

[137] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria et W. 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, Genova, Italy, April 2001. Springer-Verlag. 1.3

[138] Moshe Y. Vardi. Büchi complementation: A forty-year saga. In *5th symposium on Atomic Level Characterizations (ALC'05)*, 2005. Invited extended abstract. 3.3.6

[139] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, Nice, France, January 2007. Springer-Verlag. Invited paper. 1.4, 2.5

[140] Moshe Y. Vardi. The Büchi complementation saga. In *Proceedings of the 17th Symposium on Theoretical Aspects of Computer Science (STACS'07)*, Aachen, Germany, February 2007. Invited paper. 3.3.6

[141] Moshe Y. Vardi et Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994. 4.1, 8.5

[142] Kimmo Varpaaniemi. *On The Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, May 1998. 2.7, 8.3

[143] Francois Vernadat, Pierre Azema, et Francois Michel. Covering step graph. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*

(ICATPN'96), volume 1091 of *Lecture Notes in Computer Science*, pages 516–535. Springer-Verlag, 1996. 8.3

[144] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56 (1–2):72–99, 1983. 4.1, 4.1.3, 4.1.3

[145] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, (110–111):119–136, 1985. 4.1, 4.1.3, 4.1.3

[146] Pierre Wolper. Constructing automata from temporal logic formulas: A tutorial. In E. Brinksma, H. Hermanns, et J.-P. Katoen, editors, *Proceedings of the FMPA 2000 summer school*, volume 2090 of *Lecture Notes in Computer Science*, pages 261–277, Nijmegen, the Netherlands, July 2000. Springer-Verlag. 4.1.1

[147] Pierre Wolper et Denis Leroy. Reliable hashing without collision detection. In C. Courcoubetis, editor, *Proceedings of the 5th Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993. 5.7

[148] Pierre Wolper, Moshe Y. Vardi, et Aravinda Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pages 185–194. IEEE Computer Society Press, 1983. Later extended and published as [141]. 4.1.1, 4.1, 4.1.1

Annexe A

Spot

Spot (*Spot Produces Our Traces*) est une bibliothèque de *model checking* écrite en C++. Elle fournit différentes briques nécessaires à la réalisation d'un *model checker* selon l'approche automate pour la vérification de formules LTL.

A.1 Bibliothèque centrée sur les TGBA

Les TGBA sont manipulés par Spot sous la forme d'une classe abstraite: `tgba`. Cette classe possède une interface qu'on pourrait réduire à deux méthodes principales:

- `get_init_state()` retourne l'état initial (contrairement à la définition 27 page 48, Spot ne supporte qu'un seul état initial ; la raison est historique et cela n'est pas contraignant car un seul état initial suffit en pratique),
- `succ_iter()` retourne un intérateur sur les successeurs de l'état passé en argument. Cet itérateur indiquera non seulement les états successeurs, mais aussi les propositions et conditions d'acceptation qui étiquettent chaque transition.

Plusieurs implémentations de la classe abstraite `tgba` sont offertes. Un TGBA peut par exemple être stocké explicitement comme un graphe, ou implicitement sous la forme d'une relation BDD.

La définition de la classe `tgba` permet aussi le calcul à la volée et des extensions tierces.

A.2 Support des calculs à la volée

Un algorithme qui produit un `tgba`, par exemple un produit synchronisé de deux automates ou un traduction de formule LTL en automate, peut être implémenté non pas comme une fonction qui construit un `tgba`, mais comme une sous-classe de `tgba`. Un objet de cette classe représente le résultat de l'algorithme, même s'il n'a pas encore été calculé. C'est lors des appels à `get_init_state()` et `succ_iter()` que les calculs s'effectueront.

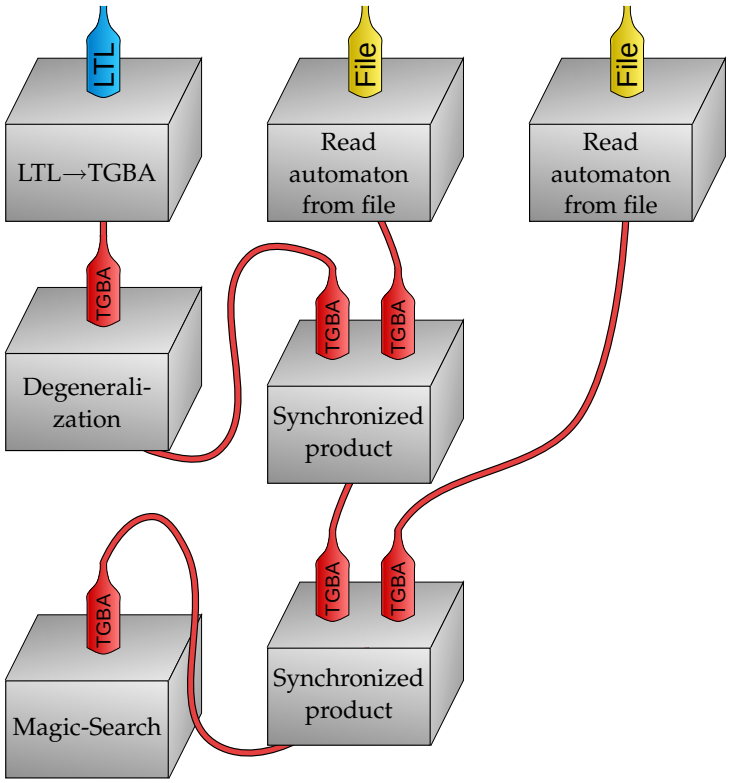


FIG. A.1: Composition de briques de *model checking*.

Implémenter des algorithmes à la volée de cette façon nous permet de les imbriquer. Par exemple nous pouvons imbriquer plusieurs produits synchronisés pour composer des modèles ou des formules, avec l'assurance que le produit global sera calculé à la volée.

Avec cette abstraction, le calcul à la volée n'entraîne aucune perte de modularité: les algorithmes sont bien découplés et peuvent être développés indépendamment.

La figure A.1 montre un exemple un peu contraint d'une telle modularité. Ici le programme possède trois entrées: une formule LTL et deux fichiers contenant des TGBA. La formule LTL est d'abord traduite en TGBA, puis cet automate est dégénéralisé, successivement synchronisé avec chacun des deux automates, puis enfin testé par un *emptiness check* (le *magic-search* de Godefroid et Holzmann [65]). Dans ce scénario, la dégénéralisation ainsi que les deux produits sont calculés à la volée, en fonction de la progression du *magic-search*.

La traduction de formule LTL n'est pas réalisée à la volée dans Spot, mais rien n'empêche de la réécrire de cette façon.

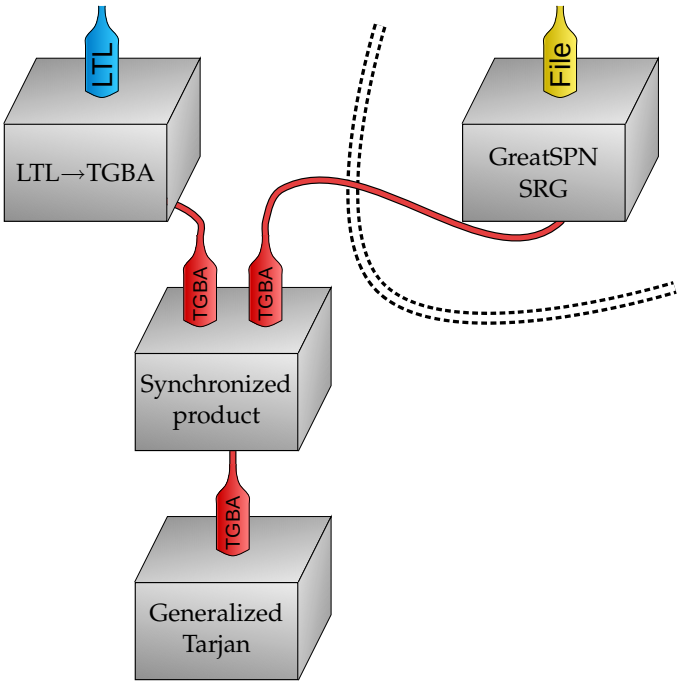


FIG. A.2: Interface entre Spot et un générateur d'espace d'état externe.

A.2.1 `tgba` utilisé comme interface pour des extensions

L'exemple précédent était artificiel parce que les automates ne sont normalement pas lus depuis des fichiers. Habituellement on voudrait que l'espace d'état soit construit à la volée à partir d'un modèle comme un réseau de Petri [41], ou un programme Promela [75].

Comme Spot ne supporte pas un tel formalisme directement, cette génération doit être effectuée en dehors de la bibliothèque. La classe `tgba` peut être vue comme une façon standard d'étendre Spot. Les utilisateurs peuvent implémenter leurs propres sous-classes de `tgba` et les utiliser de façon transparente avec les algorithmes de Spot.

La figure A.2 illustre la façon dont nous avons connecté Spot à GreatSPN[1]. GreatSPN peut produire un graphe des marquages symboliques (SRG) à partir d'un réseau de Petri coloré en exploitant ses symétries [28]. Nous avons créé une sous classe de `tgba` dont les méthodes délèguent leur travail aux procédures correspondantes de GreatSPN. Du point de vue de Spot, le SRG apparaît comme n'importe quel autre `tgba`.

La figure A.2 montre aussi une implémentation concrète de l'approche automate du *model checking* que nous avions décrite section 2.8 page 28. La boîte *Generalized Tarjan* correspond à l'algorithme présenté figure 5.8 page 121.

De façon grossière, SRG exploite les symétries globales d'un système et regroupe les éléments du système qui ne sont pas distingués par la formule [126]. Une autre approche, développée par Baarir et al. [7] ne considère pas la formule dans son ensemble, mais

[1] `http://www.di.unito.it/~greatspn/`

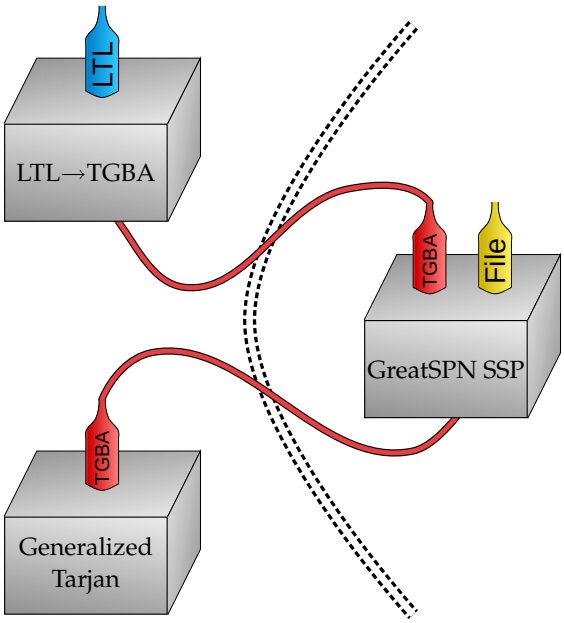


FIG. A.3: Interface avec un produit synchronisé externe.

utilise sa traduction en automate pour adapter localement les classes d'équivalence, pendant la synchronisation. Cette seconde approche, appelée *produit synchronisé symbolique* a été réalisée dans Spot comme montré par la figure A.3.

Comme on le voit sur cet exemple, ici la classe externe à Spot est plus qu'un générateur d'espace d'état, mais plutôt une sorte de produit synchronisé.

A.3 `tgba` comme un formalisme d'entrée

Il nous semble important que le formalisme d'entrée de Spot soit aussi bas niveau qu'un `tgba`. Nous espérons qu'ainsi différents formalismes de plus haut niveaux pourront être connectés. Cette remarque est valable aussi bien pour les modèles que pour les formules.

Rappelons aussi que les TGBA peuvent être vus comme une sur-classe des autres automates de Büchi ; non pas au niveau de l'expressivité (qui est identique), mais à celui de la concision. Un automate de Büchi avec étiquettes sur les états peut être transformé en un TGBA sans en changer la taille (proposition 3 page 51) alors que l'inverse n'est pas vrai. Cela signifie que Spot peut réutiliser n'importe quel algorithme produisant des automates de Büchi, où que soient leurs étiquettes. L'utilisation d'algorithmes qui attendent un automate étiqueté sur les états en entrée demandera en revanche quelques modifications (généralement mineures) pour accepter des TGBA.

A.4 Disponibilité et utilisations connues

Les sources de la bibliothèque totalisent à peu près 70000 lignes, écrites au cours des quatre dernières années.

Plusieurs autres personnes y ont contribué:
– Rachid Rebiha a aidé à sa naissance,
– Thomas Martinez a ajouté plusieurs techniques de réductions de formules LTL et d'automates à l'occasion d'un stage de DEA,
– Denis Poitrenaud y a implémenté plusieurs algorithmes d'*emptiness check* pour le plaisir,
– Heikki Tauriainen est venu ajouter une heuristique dans notre implémentation de son *emptiness check*,
– la présence de Souheib Baarir a été indispensable lors de l'implémentation de l'algorithme d'*emptiness check* avec inclusion du chapitre 6.

La bibliothèque est disponible sous licence GNU GPL depuis `http://spot.lip6.fr`. Ce site propose aussi une interface en ligne vers les algorithmes de traduction de formules LTL.

À notre connaissance, la bibliothèque a été utilisée par Sebastiani et al. [110] pour traduire des formules LTL (voir aussi section 7.4 page 166). Elle a aussi été utilisée par Tauriainen [123] et Li et al. [91] pour tester des variantes d'*emptiness check*.

Résumé

L'approche automate pour le *model checking* de propriétés temporelles à temps linéaire est une technique classique de vérification formelle de systèmes concurrents. Un système, ainsi qu'une propriété qu'on souhaite y vérifier, sont modélisés sous forme d'automates reconnaissant des mots infinis (des ω-automates). Des manipulations de ces automates permettent d'établir si le système vérifie la propriété ou non.

Dans cette thèse nous nous focalisons sur un type particulier d'ω-automates: les automates de Büchi généralisés basés sur les transitions (TGBA). Dans un premier temps nous revisitons les deux principales étapes de l'approche: la traduction de formules de logique temporelle à temps linéaire en TGBA et le test de vacuité (*emptiness check*) d'un TGBA. Pour chacune, nous proposons des améliorations des algorithmes existants, et comparons ces algorithmes pour montrer l'intérêt des TGBA.

Dans un deuxième temps, nous proposons deux variations du test de vacuité. L'une peut être utilisée avec certains algorithmes qui réduisent l'automate représentant le système en regroupant ses états de façon symbolique. Elle utilise des tests d'inclusion entre ces regroupements d'états pour guider la construction à la volée de l'automate. L'autre est une généralisation aux automates de Streett (à nouveau basés sur les transitions) ; elle permet de prendre en compte des hypothèses d'équité forte lors de la vérification.

Mot clefs: vérification formelle, *model checking*, ω-automates, automates de Büchi, logique temporelle, LTL, test de vacuité, algorithmes

Abstract

The automata theoretic approach to model checking linear-time temporal properties is a classical verification technique for concurrent systems. Both the system and the property to verify are expressed as automata on infinite words (ω-automata). Some operations on these automata establish whether the property holds in the system.

In this thesis we focus on a kind of ω-automata called *Transition-based Generalized Büchi Automata* (TGBA). We start by revisiting the main two steps of the approach: the translation of linear-time temporal logic formulæ into TGBA, and the emptiness check of TGBA. For each of these steps, we offer improvements to existing algorithms, and compare them to show the benefits of using TGBA.

We then introduce two variants of the emptiness check algorithm. The first one can be combined with algorithms that aim at reducing the automaton which represents the system by gathering some of its states symbolically. The new emptiness check uses inclusion checks between these symbolic sets of states to drive the on-the-fly construction of the automaton. The second variant is a generalization to (transition-based) Streett automata; it enables the verification of properties under strong fairness assumptions.

Keywords: formal verification, model checking, ω-automata, Büchi automata, temporal logic, LTL, emptiness check, algorithms