SPOT : une bibliothèque de vérification de propriétés de logique temporelle à temps linéaire

—

Mémoire de DEA

Alexandre DURET-LUTZ
aduret@src.lip6.fr
DEA Systèmes Informatiques Répartis, LIP6, Paris.

Rachid REBIHA
rebiha@src.lip6.fr
DEA Informatique Fondamentale, LIFO, Orléans.

Encadrants : Jean-Michel Ilié et Denis Poitrenaud, SRC/LIP6, Paris.

Septembre 2003

Id: rapstage.tex,v 1.133 2003/09/30 11:18:26 aduret Exp

Rachid REBIHA a défendu son stage au Laboratoire d'Informatique Fondamentale d'Orléan (LIFO) le 3 septembre 2003 devant un jury composé de
– Siva ANANTHARAMAN (LIFO)
– Jean-Michel ILIÉ (LIP6)
– Gaëtan HAINS (LIFO)
– Denis POITRENAUD (LIP6)
– Henri THUILLIER (LIFO)

Alexandre DURET-LUTZ a défendu son stage au Laboratoire d'Informatique de Paris 6 le 12 septembre 2003 devant un jury composé de
– Jean-Michel COUVREUR (LSV, LaBRI)
– Jean-Michel ILIÉ (LIP6)
– Fabrice KORDON (LIP6)
– Jean-François PEYRE (CNAM)
– Denis POITRENAUD (LIP6)

Nous tenons à remercier Denis POITRENAUD, Jean-Michel ILIÉ, et Akim DEMAILLE pour leurs relectures et commentaires des multiples versions de ce rapport.

D'autre part Spot ne serait pas ce qu'il est sans nos nombreuses discussions avec Denis POITRENAUD, Jean-Michel ILIÉ et Jean-Michel COUVREUR, et sans le travail de Souheib BAARIR et Yann THIERRY-MIEG sur l'interface avec GreatSPN.

Table des matières

Chapitre 1

Introduction

1.1 Vérification de systèmes

Les méthodes formelles [16] sont des langages, techniques, et outils mathématiques permettant de spécifier et vérifier des systèmes. La spécification consiste à décrire le comportement et les propriétés d'un système dans un ou plusieurs langages mathématiques. Il existe grossièrement deux types de méthodes formelles pour vérifier qu'un système est correct par rapport à sa spécification : la preuve de théorème, ou l'exploration de l'espace des états du systèmes.

Pour faire de la preuve de théorème, le système et les propriétés à prouver sont tous les deux représentés 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 système.

Tout le problème est d'automatiser ces constructions de preuves. En pratique, les outils de preuve de théorèmes sont rarement entièrement automatiques (ils requièrent l'aide interactive de l'utilisateur), c'est pourquoi leur emploi demande une bonne expertise. Quoiqu'il en soit, si ces preuves permettent de montrer la correction d'un système par rapport à sa spécification, elles se révèlent peu pratiques en cas d'erreur : si une propriété du système n'est pas de celui-ci, il est difficile de trouver le comportement du système qui en est la cause.

Les méthodes basées sur l'exploration de l'espace d'états, désignées aussi sous le nom de *model checking* (vérification de modèle) procèdent différemment [51]. Ces méthodes reposent sur une modélisation du système appelée espace d'états, décrivant tous les états accessibles du système ainsi que les liens entre eux. Une telle structure est immense, mais peut être construite automatiquement à partir d'une spécification de plus haut niveau. Par exemple l'espace d'états d'un système modélisé sous la forme d'un réseau de Petri correspond au graphe des marquages accessibles de celui-ci.

De nombreuses propriétés peuvent alors être vérifiées algorithmiquement sur cet espace d'états. Le travail de l'utilisateur consiste à poser les bonnes questions — 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 propriété 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 pouvant atteindre un nombre fini de configurations. Par exemple les circuits logiques et certains protocoles de communications sont des systèmes à états finis. Cette restriction est due au fait que ces méthodes procèdent par exploration de l'ensemble des états accessibles.

Les logiciels ne possèdent pas, pour la plupart, un ensemble d'états fini. C'est le cas par exemple si leur consommation mémoire n'est pas bornée, s'ils gèrent des files non-bornées de messages asynchrones, si des processus se répliquent, etc. Nous ne nous intéresserons ici qu'au *model checking* de systèmes finis par la suite. De nombreuses techniques essayent de ramener un système infini à une abstraction finie. Esparza [24] donne une courte bibliographie sur la vérification de systèmes dont l'ensemble d'états est infini.

Le principal problème rencontré par un *model checker* est la taille de l'espace d'états à vérifier. En effet, l'ensemble des états à couvrir est souvent trop grand pour tenir dans la mémoire d'un ordinateur décent. Un second problème concerne la vérification des propriétés sur cet espace. Plusieurs formalismes, d'expressivité variable, existent pour exprimer ces propriétés. En général le gain en expressivité entraîne une complexité plus grande de l'algorithme de vérification, donc un temps de calcul plus lent.

La quête du Graal, en *model checking*, est donc double : on y cherche une représentation compacte de l'espace d'états, ainsi qu'un formalisme suffisamment expressif et vérifiable efficacement [66].

CTL et LTL 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 l'on considère l'ensemble des séquences d'exécution possibles du système.

Les expressivités de CTL et LTL ne sont pas comparables [71]. 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 (on vérifie une propriété sur toutes les exécutions possibles de processus), et permet d'exprimer directement certaines propriétés importantes comme l'équité. D'autre part, contrairement à celui utilisant LTL, le *model checking* utilisant CTL n'exhibe pas de contre-exemple sous la forme de séquences 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).

La vérification d'une formule LTL étant une opération de complexité exponentielle par rapport à la taille de la formule, de nouveaux algorithmes — censément plus performants les uns que les autres — sont publiés régulièrement.

1.2 Objectifs du stage

A l'origine l'équipe SRC du Lip6 souhaitait se munir d'un outil de vérification de réseau de Petri coloré. Cet outil devait répondre à plusieurs critères :
- Il devait être écrit en C++.
- Il devait s'interfacer avec GreatSPN (un outil d'analyse de réseaux de Petri bien formés développé à Turin [13]) et utiliser ce dernier pour explorer le graphe des marquages accessibles afin de tirer parti des différentes techniques de réduction de l'espace d'état implémentées. Plusieurs personnes de l'équipe SRC travaillent sur ces techniques et cet outil.
- Il devait être conçu de façon modulaire pour faciliter l'intégration et la comparaison des nouvelles techniques de *model checking* tant sur la modélisation du système que sur les avancées algorithmiques de vérification.
- Il devait intégrer plusieurs de ces techniques et les comparer.
- Parmi ces dernières avancées, il devait notamment intégrer l'approche de Couvreur [20].
- Il devait prendre en compte des techniques de réduction du graphe des marquages et de l'automate.

Naturellement — comme tout sujet de stage qui se respecte — ces objectifs ont un peu évolué au cours du stage. L'outil a pris la forme d'une bibliothèque de *model checking*, et l'accent a beaucoup porté sur l'organisation de celle-ci : la volonté étant celle de fournir des briques de *model checking*, qu'un utilisateur pourrait combiner facilement pour construire son propre *model checker*. La réalisation d'une architecture modulaire qui n'empêche pas les optimisations n'a pu se faire qu'après une réflexion sur les interfaces des différents modules vis-à-vis des optimisations souhaitées.

La nécessité de s'interfacer avec GreatSPN a limité le domaine de travail de la bibliothèque. Cette dernière n'a en effet pas besoin de savoir ce qu'est un réseau de Petri puisque GreatSPN lui fournit un graphe des marquages, et elle n'a pas non plus besoin de chercher à réduire des graphes. Les points à développer concernent donc les autres aspects du *model checking* :
- la traduction d'une formule LTL en un automate de Büchi,
- le produit synchronisé d'automates,
- et la recherche de contre-exemples.

Enfin la possibilité d'utiliser cette bibliothèque à partir de scripts est apparue importante pour faciliter les expérimentations.

1.3 Organisation de ce mémoire

Ce document se compose de deux grosses parties.

Le chapitre 2 est une étude bibliographique de la vérification de propriétés de logique temporelle à temps linéaire. Ce chapitre introduit l'approche automate utilisée pour la vérification, et présente les différents algorithmes existants dont certains seront implémentés dans la bibliothèque. Cette étude, bien qu'indépendante de la bibliothèque permet de justifier des choix qui y ont été faits, tels que l'utilisation d'un type particulier d'automates.

Les chapitres 3, 4, et 5 décrivent la bibliothèque développée. Le chapitre 3 présente formellement les automates qui seront au centre de la bibliothèque. Le chapitre 4 détaille l'utilisation et le fonctionnement de chacun des modules de la bibliothèque. Enfin le chapitre 5 discute de la façon dont la bibliothèque a été testée, et de ce qu'il reste à faire.

Chapitre 2

Approche automate de la vérification de propriétés de Logique Temporelle à temps Linéaire

Ce chapitre tente de présenter les différents algorithmes et techniques employés pour la vérification de formules LTL. Il est organisé comme suit. La première section pose quelques définitions et présente le schéma général de l'approche automate, qui constitue le cadre de ce rapport. La section 2.2 définit les différents automates fréquemment rencontrés en *model checking*. Les sections 2.3 et 2.4 décrivent les algorithmes centraux en *model checking* : la traduction d'une formule en un automate, et l'*emptiness check* d'un automate. Les sections suivantes abordent quelques points plus annexes : prise en compte d'hypothèses d'équité, techniques d'implémentations, techniques de vérification des traducteurs de formule LTL. Une dernière section présente un *model checker* incontournable, SPIN.

2.1 Préliminaires

Dans cette section, nous introduisons différents concepts utilisés en *model checking*. Les structures de Kripke permettent de modéliser les comportements d'un système. La logique temporelle peut exprimer des propriétés sur ces comportements. Les ω-automates, nous servent à modéliser des systèmes *et* des formules de LTL de façon unifiée.

2.1.1 Propositions atomiques

Nous représentons les états d'un système à tester comme la valuation d'un ensemble fini de variables propositionnelles. 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. Dans la pratique ces variables sont associées à des propriétés de plus haut niveau concernant le système ou son modèle. Par exemple on pourrait avoir une variable $demande$ qui indique que le système effectue une demande, et une variable $réponse$ qui indique la réception d'une réponse.

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 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 : $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, $\neg p \wedge \neg q$.

2^{AP} désigne l'ensemble des parties de AP. Dans notre exemple $2^{AP} = \{\{p, q\}, \{p\}, \{q\}, \emptyset\}$. Il existe une bijection entre 2^{AP} et l'ensemble des *minterm* sur AP : il suffit d'interpréter les éléments de 2^{AP} comme des listes de propositions positives, toutes les autres propositions étant niées.

$$\begin{aligned} \{p, q\} &\leftrightarrow p \wedge q \\ \{p\} &\leftrightarrow p \wedge \neg q \\ \{q\} &\leftrightarrow \neg p \wedge q \\ \emptyset &\leftrightarrow \neg p \wedge \neg q \end{aligned}$$

Nous utiliserons l'une ou l'autre de ces représentations de façon équivalente[1], et nous noterons 2^{AP} l'ensemble des *minterms* de propositions atomiques.

De façon similaire, nous désignerons par $2^{2^{AP}}$ l'ensemble des formules propositionnelles sur AP, car il existe une bijection entre ces deux ensembles. Chaque élément de $2^{2^{AP}}$, par exemple $\{\{p\}, \emptyset\}$, peut en effet être interprété comme une disjonction de *minterms*, par exemple $(p \wedge \neg q) \vee (\neg p \wedge \neg q)$, et d'autre part tout formule propositionnelle peut-être exprimée sous cette forme. Ainsi nous pouvons assimiler la formule $\neg q$ et l'ensemble $\{\{p\}, \emptyset\}$.

2.1.2 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 états du système.

Définition 1. *Une structure de Kripke est un quintuplet $KS = \langle AP, S, \delta, l, s_0 \rangle$ où*
- *AP est une ensemble fini de propositions atomiques,*
- *S est un ensemble fini d'états (les nœuds du graphe),*
- *$\delta : S \mapsto 2^S \setminus \{\emptyset\}$ est une fonction indiquant les états successeurs d'un état,*
- *$l : S \mapsto 2^{AP}$ est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état,*
- *s_0 est l'état initial.*

Une exécution du système est une séquence infinie d'états de KS, c'est-à-dire une fonction $\sigma : \mathbb{N} \mapsto S$ telle que $\sigma(0) = s_0$ et $\forall i \in \mathbb{N}, \sigma(i+1) \in \delta(\sigma(i))$. La définition de δ assure qu'une exécution existe. (Si le système contient un état s sans successeur, il suffit de poser $\delta(s) = \{s\}$ pour assurer le caractère infini des exécutions.)

Pour une exécution σ, nous noterons $\pi = l(\sigma)$ la séquence définie par $\forall i \geq 0, \pi(i) = l(\sigma(i))$. π est le mot généré par σ. Comme ce mot est de longueur infinie, on dit aussi ω-mot.

2.1.3 Structure de Kripke séquentielle

Une exécution dans une structure de Kripke est une séquence infinie. Or une structure de Kripke est un graphe fini. Il existe donc nécessairement un point à partir duquel la séquence boucle dans la structure de Kripke.

Une exécution peut être vue comme une structure de Kripke dite séquentielle. La figure 2.1 représente une telle structure.

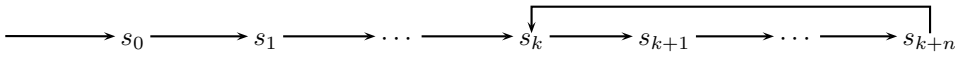


FIG. 2.1 – Une structure de Kripke séquentielle.

Une particularité des structures de Kripke séquentielles est que les deux logiques CTL et LTL y sont également expressives. Ce point sera utile section 2.7.

2.1.4 Structure de Kripke étiquetée

Une structure de Kripke peut être étiquetée sur les transitions pour représenter les événements dans un système. De tels événements peuvent par exemple correspondre aux données reçues par le programme.

Définition 2. *Une structure de Kripke étiquetée est un sextuplet $KS = \langle AP, E, S, \delta, l, s_0 \rangle$ où*
- *AP est un ensemble fini de propositions atomiques,*
- *E est un ensemble fini d'événements,*
- *S est un ensemble fini d'états,*
- *$\delta : S \mapsto 2^{E \times S} \setminus \{\emptyset\}$ est une fonction indiquant les états successeurs d'un état ainsi que les événements associés aux transitions.*
- *$l : S \mapsto 2^{AP}$ est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état,*
- *s_0 est l'état initial.*

La version déterministe et complète d'une telle structure est appelée machine de Mealy.

Définition 3. *Une machine de Mealy est un sextuplet $KS = \langle AP, E, S, \delta, l, s_0 \rangle$ où*
- *AP est un ensemble fini de propositions atomiques,*

[1] L'utilisation de la notation ensembliste est en principalement dictée par des contraintes techniques ; par exemple dans des figures réalisées avec des outils n'offrant pas de symboles mathématiques tels que $\wedge$ ou $\vee$.

– E *est un ensemble fini d'événements,*
– S *est un ensemble fini d'états,*
– $\delta : S \times E \mapsto S$ *est une fonction indiquant l'état successeur pour une paire (état, événement) donnée,*
– $l : S \mapsto 2^{AP}$ *est une fonction indiquant l'ensemble des propositions atomiques satisfaites dans un état,*
– s_0 *est l'état initial.*

2.1.5 ω-automates

Un ω-automate est un automate *fini* reconnaissant des mots de longueur *infinie* (ou ω-mots) [34].

Il a la même structure qu'un automate classique : des états, des transitions entre ces états, des étiquettes choisies dans l'alphabet des mots à reconnaître, des états initiaux. La seule différence porte sur l'absence d'états finaux (ou états d'acceptation). En effet, de tels états n'ont de sens que lorsque les mots à reconnaître sont finis. Dans le cas des mots infini, qui par définition en se terminent pas, on généralisera les états d'acceptation en des conditions d'acceptation.

Un ω-mot est reconnu par un ω-automate s'il existe un chemin dans l'automate étiqueté par les lettres du mots, et il est accepté si ce chemin vérifie les conditions d'acceptation de l'automate.

Nous reviendrons sur les différentes façons de définir ces conditions d'acceptation section 2.2. Pour fixer les idées, un exemple de condition d'acceptation peut être « le chemin passe infiniment souvent par l'état s ».

Définition 4. *De façon générale, un ω-automate étiqueté sur les transitions sur l'alphabet AP est un quintuplet $A = \langle AP, \mathcal{Q}, \delta, \mathcal{I}, \mathcal{F} \rangle$ où*
– AP *est un ensemble fini de propositions atomiques,*
– $\mathcal{Q}$ *est ensemble fini d'états,*
– $\delta : \mathcal{Q} \to 2^{2^{AP} \times \mathcal{Q}}$ *est la fonction de transition de l'automate, chaque transition étant étiquetée par une formule propositionnelle,*
– $\mathcal{I} \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
– $\mathcal{F}$ *est un ensemble associé aux conditions d'acceptation.*

La définition précise de $\mathcal{F}$ et son interprétation dépendent du type ω-automate considéré.

Comme dans les structures de Kripke, chaque séquence infinie σ de l'automate est associée à un ω-mot π. Ici les lettres de π sont les étiquettes des transitions franchies. Nous noterons $\mathscr{L}(A)$ le langage de A, c'est-à-dire l'ensemble des mots associés aux séquences infinies acceptées par l'automate.

Définition 5. *Un ω-automate étiqueté sur les états est un ω-automate étiqueté sur les transitions dans lequel les transitions sortantes d'un même état portent toutes la même étiquette.*

Comme son nom l'indique, un tel automate est traditionnellement représenté en faisant porter à l'état l'étiquette commune à toutes les transitions sortantes. Il peut être représenté par un sextuplet $A = \langle AP, \mathcal{Q}, \delta, l, \mathcal{I}, \mathcal{F} \rangle$ où
– AP *est un ensemble fini de propositions atomiques,*
– $\mathcal{Q}$ *est ensemble fini d'états,*
– $\delta : \mathcal{Q} \to 2^{\mathcal{Q}}$ *est la fonction de transition de l'automate,*
– $l : \mathcal{Q} \to 2^{2^{AP}}$ *est une fonction indiquant l'étiquette de chaque état,*
– $\mathcal{I} \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
– $\mathcal{F}$ *est un ensemble associé aux conditions d'acceptation.*

Il existe plusieurs types d'ω-automates [65]. Leurs différences tiennent à la façon dont sont définies les conditions d'acceptation. Les plus connus sont les automates de Büchi, introduits par Büchi [9] comme moyen de montrer la satisfaction d'une formule de logique du second ordre. Ce sont aussi les plus employés en *model checking*. Dans un automate de Büchi $\mathcal{F}$ est un ensemble d'états. Une séquence infinie σ de A est acceptée si elle passe infiniment souvent par des états de $\mathcal{F}$.

Il faut noter que la définition d'ω-automate donnée ci-dessus correspond à un automate non-déterministe (en effet plusieurs transitions sortantes d'un même état peuvent avoir la même étiquette, d'autre part la définition supporte plusieurs états initiaux). À la différence d'un automate classique, un ω-automate quelconque ne peut *pas* être systématiquement déterminisé [70, 65].

Les différents types d'ω-automates utilisés en *model checking* seront présentés plus en détail section 2.2.

Une opération sur les automates importante en *model checking* est le produit synchronisé de deux automates. Cette opération permet de calculer l'intersection des langages de deux automates, c'est-à-dire leurs comportements communs.

Définition 6. *Le produit synchronisé de deux automates $A_1 = \langle AP, \mathcal{Q}_1, \delta_1, \mathcal{I}_1, \mathcal{F}_2 \rangle$ et $A_2 = \langle AP, \mathcal{Q}_2, \delta_2, \mathcal{I}_2, \mathcal{F}_2 \rangle$ est l'automate $A_1 \otimes A_2 = \langle AP, \mathcal{Q}_3, \delta_3, \mathcal{I}_3, \mathcal{F}_3 \rangle$ défini par*

– $\mathcal{Q}_3 = \mathcal{Q}_1 \times \mathcal{Q}_2$
– $\delta_3((q_1, q_2)) = \{((L_1 \cap L_2), (s_1, s_2)) \,|\, (L_1, s_1) \in \delta_1(q1) \wedge (L_2, s_2) \in \delta_2(q2) \wedge L_1 \cap L_2 \neq \emptyset\}$
– $\mathcal{I}_3 = \mathcal{I}_1 \times \mathcal{I}_2$
– $\mathcal{F}_3$ *est défini de façon ad hoc en fonction de* $\mathcal{F}_1$ *et* $\mathcal{F}_2$ *et du type de l'automate.*
Alors $\mathscr{L}(A_1 \otimes A_2) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$.

2.1.6 LTL : logique temporelle à temps linéaire

Un état dans une structure de Kripke traduit l'état du système à un instant donné. Les successeurs de cet état indiquent les états futurs possibles, tandis que les prédécesseurs sont les états passés qui ont pu mener à ce présent.

Un état du système étant étiqueté par des propositions de AP, il est possible d'exprimer une propriété sur cet état avec une formule de logique propositionnelle. En revanche, la logique propositionnelle ne suffit pas à exprimer des propriétés concernant les variations des étiquettes au fil d'une exécution. Par exemple, nous souhaiterions exprimer des propriétés telles que « *un état dans lequel une demande est effectuée est toujours suivi d'un état dans lequel une réponse est reçue* », autrement dit, « *dans toute séquence d'exécution, tout état étiqueté par* demande *est suivi d'un état étiqueté par* réponse »[2]. De façon très approximative, de telles propriétés, commençant par « *dans toute séquence d'exécution* », peuvent pour la plupart s'exprimer en logique temporelle à temps linéaire (LTL).

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*).
Les règles de formation d'une formule sont les suivantes.
– Une proposition atomique $p \in AP$ est une formule.
– Si f_1 et f_2 sont deux formules, alors $f_1 \wedge f_2$, $\neg f_1$, $\mathsf{X}\, f_1$, et $f_1 \,\mathsf{U}\, f_2$ sont aussi des formules.
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*) et G (*always*) sont aussi définis en tant qu'abréviations : $\mathsf{F}\, f = \top \,\mathsf{U}\, f$ et $\mathsf{G}\, f = \neg \mathsf{F} \neg f = \neg(\top \,\mathsf{U}\, \neg f)$.

Cette séparation entre opérateurs de base et abréviations est utile pour simplifier les démonstrations. Au niveau algorithmique les opérateurs F et G peuvent aussi être considérés comme des abréviations. Mais pour des raisons pratiques on introduit souvent l'opérateur dual de U, noté ici[3] R (*release*) : $f_1 \,\mathsf{R}\, f_2 = \neg(\neg f_1 \,\mathsf{U}\, \neg f_2)$. Cela permet de supprimer bon nombre de négations lors de réécritures de la formule. Par exemple G peut être redéfini comme $\mathsf{G}\, f = \bot \,\mathsf{R}\, f$.

L'interprétation d'une formule LTL f se fait par rapport à une exécution infinie σ dans une structure de Kripke $KS = \langle S, \delta, l, s_0 \rangle$. Nous noterons $\sigma \models f$ lorsque σ satisfait la formule f, et $KS \models f$ lorsque toute exécution infinie de KS satisfait f. Pour une proposition atomique p, et deux formules logiques f_1 et f_2, la satisfaction d'une formule LTL par rapport à σ est définie inductivement de la façon suivante.

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

où $\sigma^i : \mathbb{N} \mapsto S$ est la fonction définie par $\sigma^i(n) = \sigma(i+n)$, qui représente le suffixe de l'exécution σ commençant à l'instant i.

Les équations suivantes se démontrent aisément à partir de la définition des opérateurs et de leur interprétation.

$$
\begin{aligned}
\mathsf{F}\, g &= g \vee \mathsf{X}\, \mathsf{F}\, g \\
\mathsf{G}\, g &= g \wedge \mathsf{X}\, \mathsf{G}\, g \\
f \,\mathsf{U}\, g &= g \vee (f \wedge \mathsf{X}(f \,\mathsf{U}\, g)) \\
f \,\mathsf{R}\, g &= g \wedge (f \vee \mathsf{X}(f \,\mathsf{R}\, g))
\end{aligned}
\tag{2.1}
$$

Ces identités sont souvent utilisées pour programmer des algorithmes récursifs : elles séparent ce qui doit être satisfait à l'instant présent de ce qui doit l'être à l'instant suivant. Cependant, il faut prendre garde aux opérateurs F et U. Les

[2] Notons que cette spécification autorise deux demandes à être suivies d'une seule réponse.
[3] Les opérateurs X, F, G et R sont aussi notés $\bigcirc$, $\diamond$, $\Box$ et V respectivement.

formules $f \mathsf{U} g$ et $\mathsf{F}\, g$ impliquent toutes les deux qu'il existe un instant du futur tel que g soit satisfaite. Appliquer récursivement $g \vee (f \wedge \mathsf{X}(f \mathsf{U} g))$ pour vérifier la satisfaction de $f \mathsf{U} g$ n'est pas suffisant, car l'application récursive de cette première formule acceptera une séquence constituée uniquement d'états satisfaisant f.

Par la suite nous dirons que *$f \mathsf{U} g$ et $\mathsf{F}\, g$ promettent* de satisfaire g. Les identités (2.1) peuvent être utilisées, à conditions de s'assurer que les séquences qu'elles permettent de reconnaître *tiennent leurs promesses*.

L'arbre de syntaxe abstraite d'une formule LTL sous forme normale négative représente la formule et ses sous-formules sous une forme arborescente, après suppression des abréviations. Les nœuds non-terminaux sont étiquetés par les opérateurs logiques utilisés et les feuilles par des littéraux.

L'exemple « *dans toute séquence d'exécution, tout état étiqueté par demande est suivi d'un état étiqueté par réponse* » s'écrirait $\mathsf{G}(demande \Rightarrow \mathsf{F}\, réponse)$. En supposant, comme dans la plupart des algorithmes qui seront présentés que R est un opérateur de base, cette formule peut ce récrire $\bot \mathsf{R}((\neg demande) \vee (\top \mathsf{U}\, réponse))$. Son arbre de syntaxe abstraite est représenté figure 2.2 (les négations étant repoussées au niveau des feuilles).

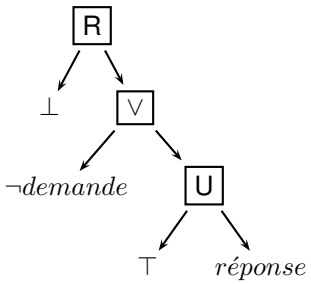


FIG. 2.2 – Arbre de syntaxe pour $\mathsf{G}(demande \Rightarrow \mathsf{F}\, réponse) = \bot \mathsf{R}((\neg demande) \vee (\top \mathsf{U}\, réponse))$.

La taille d'une formule f, notée $|f|$, est le nombre de nœuds de son arbre syntaxique. La complexité d'un algorithme travaillant sur une formule logique est souvent donnée par rapport à sa taille.

Nous ne nous intéressons ici qu'au fragment *futur* de LTL. Il est possible de rajouter à LTL des opérateurs relatifs au passé (P pour *previous* et S pour *since*). La logique composée des fragments futurs et passés n'est pas plus expressive que celle constituée uniquement du fragment futur [50], mais elle rend les spécifications plus naturelles. (Par exemple il devient plus facile de traduire une contrainte telle que « *l'alarme n'est activée que s'il s'est produit un incident* ».) Il existe des algorithmes pour convertir une formule LTL faisant apparaître des opérateurs du passé en une formule LTL n'utilisant que le fragment *futur*, cette conversion peut, dans le pire des cas, produire une formule dont la taille est exponentielle par rapport à la formule originale [47].

2.1.7 Méthode des tableaux pour la satisfaction d'une formule LTL

Un tableau sémantique est une façon de prouver qu'une formule de logique propositionnelle $f(p_0, p_1, \ldots, p_n)$ est satisfiable[4]. Le plus souvent un tableau est utilisé pour des raisonnements par réfutation, c'est-à-dire que pour prouver que f est une tautologie[5], 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 satisfaisabilité d'une feuille implique celle de la formule.

Des règles de tableau indiquent comment construire le tableau à partir de la formule. Par exemple le tableau[6] 2.1 donne celles utilisées en logique propositionnelle.

Une preuve par tableau s'effectue en partant de la formule f dont on veut établir la satisfaisabilité, 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 $\neg p$, 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 (dans ce cas, $\neg f$ est un théorème).

[4] C'est-à-dire qu'il existe une affectation A des variables propositionnelles $p_0, p_1, \ldots, p_n$ telle que $f(A(p_0), A(p_1), \ldots, A(p_n)) = \top$.
[5] $f(A(p_0), A(p_1), \ldots, A(p_n)) = \top$ pour toute affectation A.
[6] 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 [4, 5] avant de prendre progressivement celle d'un arbre [59].

	formule	1er fils	2e fils
(1)	$\neg\neg f$	$\{f\}$	
(2)	$\neg\top$	$\{\bot\}$	
(3)	$\neg\bot$	$\{\top\}$	
(4)	$f \wedge g$	$\{f, g\}$	
(5)	$f \vee g$	$\{f\}$	$\{g\}$
(6)	$\neg(f \wedge g)$	$\{\neg f\}$	$\{\neg g\}$
(7)	$\neg(f \vee g)$	$\{\neg f, \neg g\}$	

TAB. 2.1 – Règles de tableau pour la logique propositionnelle.

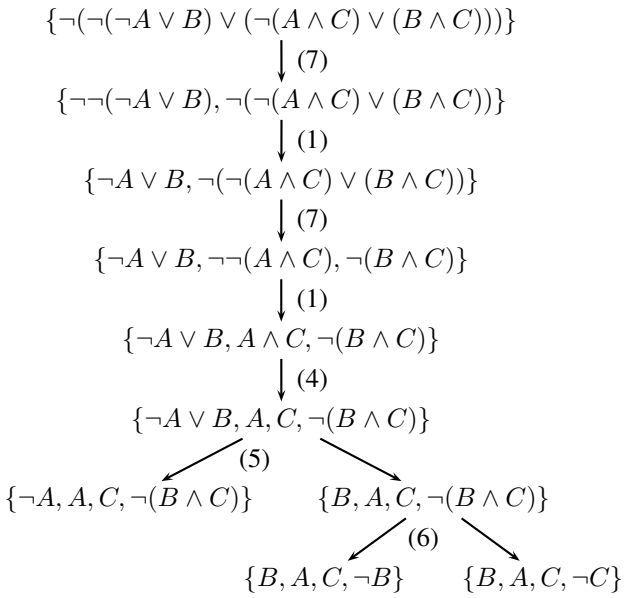


FIG. 2.3 – Preuve de $\neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C))$ par tableau en appliquant successivement les règles (7), (1), (7), (1), (4), (5) et (6) du tableau 2.1.

La figure 2.3 démontre que $f(A, B, C) = \neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C))$[7] est une tautologie en utilisant un tableau. La racine de l'arbre est $\neg f$, 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.

Fitting [26] donne la preuve de correction d'une démonstration par tableau, ainsi qu'une implémentation pour la logique propositionnelle.

Ben-Ari et al. [3] étendent ce schéma de preuve aux logiques à temps arborescent. Tandis que Wolper [75, 76] 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 : les sous-formules à vérifier dans l'état présent, et celles à vérifier dans le futur. Cette seconde partie est introduite par l'opérateur X. Les opérateurs comme F, G, U et R peuvent être récrits de façon à faire apparaître X grâce aux équations (2.1) (page 10). Par exemple le tableau 2.2 présente les règles de tableaux pour les opérateurs X et U.

formule	1er fils	2e fils
$\neg\mathsf{X} f$	$\{\mathsf{X}\neg f\}$	
$f\,\mathsf{U}\,g$	$\{g\}$	$\{f, \mathsf{X}(f\,\mathsf{U}\,g)\}$
$\neg(f\,\mathsf{U}\,g)$	$\{\neg f, \neg g\}$	$\{\neg g, \neg\mathsf{X}(f\,\mathsf{U}\,g)\}$

TAB. 2.2 – Règles de tableau supplémentaires pour la logique temporelle linéaire. Nous n'avons défini que la réécriture des opérateurs X et U. Les règles de réécriture pour F, G, et R et leurs négations peuvent être déduites des équations (2.1) de façon similaire.

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

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 $\{\bot\}$. 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 en strates du tableau. Chaque strate correspond à un instant.
– Les équations (2.1) (page 10) 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 états identiques de l'arbre.
– Trouver une branche 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 (2.1) ne garantissent pas. Par exemple le tableau va pouvoir boucler infiniment autour de l'état $\{f, \mathsf{X}(f \,\mathsf{U}\, g)\}$ dans une branche qui ne satisfait jamais g. Ces cas seront détectés dans une seconde étape d'analyse effectuée après la construction du tableau[8]. Cette seconde étape a besoin de connaître les promesses associées à chaque état, c'est-à-dire la liste de formules du type $\mathsf{F}\, b$ ou $a \,\mathsf{U}\, b$. Aussi, à la différence des tableaux en logique propositionnelle, on ne retirera pas les formules récrites des étiquettes des nœuds, on se contentera juste de les isoler pour ne pas les récrire à nouveau.

La figure 2.4 illustre la construction du tableau pour $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$. Les états sont étiquetés par des couples d'ensembles représentés sous forme de fractions $\frac{D}{F}$ où F est l'ensemble des formules à récrire (si possible) et D est l'ensemble des formules déjà récrites. (D pour *done*, et F pour *formulae*).

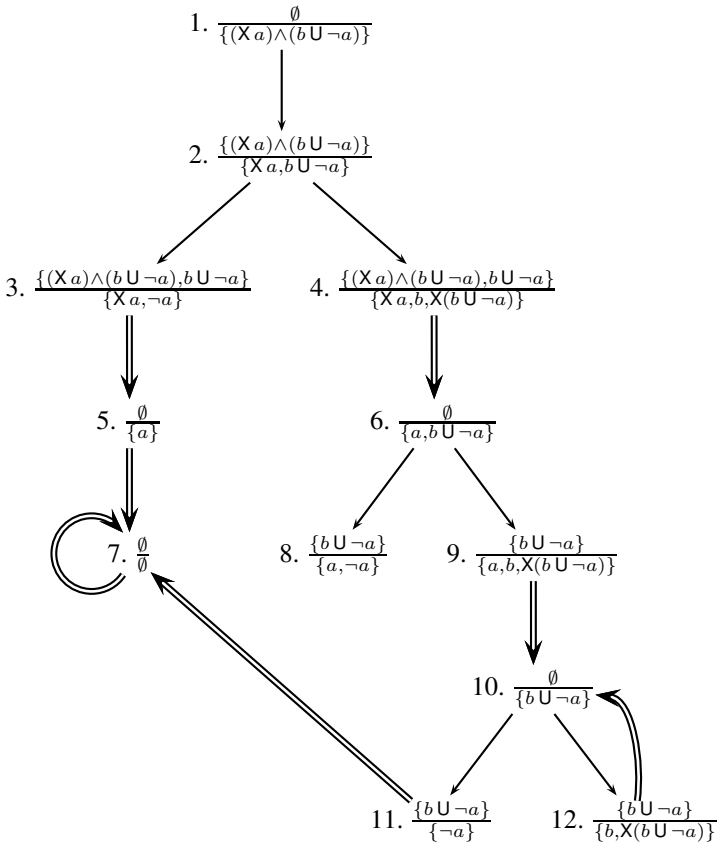


FIG. 2.4 – Tableau pour $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$. Les doubles flèches marquent le passage à un instant futur. Le nœud 8 est une contradiction.

Un nœud du graphe qui ne contient que des formules atomiques, leur négation, ou des formules précédées par X, est appelé un état. La racine du tableau et les successeurs immédiats des états sont appelés des pré-états. Intuitivement, si l'on découpe le tableau de la figure 2.4 au niveau des transitions marquées d'une double flèche, on obtient une collection de sous-tableaux comparables à ceux obtenus en logique propositionnelle (un seul instant est considéré).

[8] Cela est semblable au passage d'un automate à un ω-automate : les états d'acceptation ont été généralisés en conditions d'acceptation.

Les états sont les feuilles de ces sous-tableaux (nœuds 3, 4, 5, 7, 9, 10, 11 et 12) et les pré-états sont les racines de ces sous-tableaux (nœuds 1, 5, 6, 7, 10).

Un nœud est invalide
– s'il contient des propositions contradictoires (p.ex., p et $\neg p$)
– ou si *tous* ses fils sont invalides
– ou si c'est un pré-état contenant une formule du type $a \mathsf{U} b$ (ou $\mathsf{F} b$) et que ni ce nœud, ni aucun de ses fils valides (directs ou indirects) ne satisfont b.

Par exemple le nœud 11 est valide, donc 10 l'est aussi. 9 est valide car d'une part 10 est valide et d'autre part le nœud 11 satisfait $\neg a$ (il tient la promesse faite par $b \mathsf{U} \neg a$).

Enfin, une formule de logique temporelle f est satisfiable si la racine de son tableau est valide [76].

Cet algorithme de décision pour la satisfaisabilité d'une formule de logique temporelle aura une influence importante sur les algorithmes de traduction de formules en ω-automate présentés par la suite (section 2.3). Intuitivement, les propositions atomiques apparaissant dans les états du tableau correspondront aux propositions vérifiées par les états ou les transitions de l'automate.

2.1.8 Schéma général de l'approche par automate

L'approche par automate du *model checking* [68, 70] consiste à interpréter le système et la formule à vérifier comme deux ω-automates, A_S et A_f, pour ramener le problème à des comparaisons de langages. En particulier, nous voudrions nous assurer que $\mathscr{L}(A_S) \subseteq \mathscr{L}(A_f)$. Une telle comparaison n'est pas facile à programmer, il faudrait par exemple énumérer les mots de chaque langage. Il est plus pratique de vérifier que $\mathscr{L}(A_S) \cap \overline{\mathscr{L}(A_f)} = \emptyset$. Comme la complémentation d'un automate de Büchi est une opération exponentielle, il est plus facile de nier la formule f avant la traduction, et de tester que $\mathscr{L}(A_S) \cap \mathscr{L}(A_{\neg f}) = \emptyset$, c'est-à-dire que le langage du produit synchronisé $A_S \otimes A_{\neg f}$ est vide (cf. section 2.1.5) :

$$\mathscr{L}(A_S \otimes A_{\neg f}) = \emptyset.$$

La figure 2.5 illustre les opérations effectuées pour vérifier qu'un système satisfait une formule. Traduire le système en un ω-automate ne demande aucun travail lorsque le système est déjà représenté par une structure de Kripke. Il existe de nombreux algorithmes de traduction de formule LTL en ω-automate, ils seront présentés section 2.3. Le problème (décidable) de savoir si le langage reconnu par un ω-automate est vide est appelé *emptiness check*. Il existe aussi de nombreux algorithmes pour décider de l'*emptiness check*, ceux-ci seront présentés section 2.4.

Vardi [68] cite deux intérêts de l'emploi d'une approche par automate. D'une part, la description des algorithmes et le raisonnement sur ces algorithmes est simple et clair (p.ex. calcul de complexité). D'autre part l'extension de la logique temporelle par de nouveaux opérateurs est facilité. La prise en compte de ces nouveaux opérateurs ne se fait que lors de la traduction de la formule en automate : la suite reste inchangée.

2.2 Bestiaire des ω-automates utilisés pour la vérification LTL

Nous décrivons ici les espèces d'ω-automates communément rencontrées en *model checking* de propriétés LTL [70]. Les sept premières partagent la définition d'ω-automate donnée section 2.1.5.

1. Un automate de Büchi étiqueté sur les états est un ω-automate avec l'ensemble des états d'acceptation $\mathcal{F} \subseteq \mathcal{Q}$. Les exécutions acceptées sont celles qui passent infiniment souvent par des états de $\mathcal{F}$.
2. Un automate de Büchi *généralisé* étiqueté sur les états est un ω-automate avec l'ensemble des ensembles d'états d'acceptation $\mathcal{F} = \{\mathcal{F}_1, ..., \mathcal{F}_n\}$ où $\mathcal{F}_i \subseteq \mathcal{Q}$. Les exécutions acceptées sont celles qui, pour tout i, passent infiniment souvent par des états de $\mathcal{F}_i$.
3. Un automate de Büchi étiqueté sur les transitions est un ω-automate avec l'ensemble des transitions d'acceptation $\mathcal{F} \subseteq \mathcal{Q} \times 2^{AP} \times \mathcal{Q}$. Les exécutions acceptées sont celles qui passent infiniment souvent par des transitions de $\mathcal{F}$.
4. Un automate de Büchi *généralisé* étiqueté sur les transitions (noté TGBA) est un ω-automate avec l'ensemble des ensembles de transitions d'acceptation $\mathcal{F} = \{\mathcal{F}_1, ..., \mathcal{F}_n\}$ où $\mathcal{F}_i \subseteq \mathcal{Q} \times 2^{AP} \times \mathcal{Q}$. Les exécutions acceptées sont celles qui, pour tout i, passent infiniment souvent par des transitions de $\mathcal{F}_i$.
5. Un automate de co-Büchi *généralisé* étiqueté sur les transitions est un ω-automate avec l'ensemble des ensembles de transitions d'acceptation $\mathcal{F} = \{\mathcal{F}_1, ..., \mathcal{F}_n\}$ où $\mathcal{F}_i \subseteq \mathcal{Q} \times 2^{AP} \times \mathcal{Q}$. Les exécutions acceptées sont celles qui, pour tout i, passent finiment souvent par des transitions de $\mathcal{F}_i$.

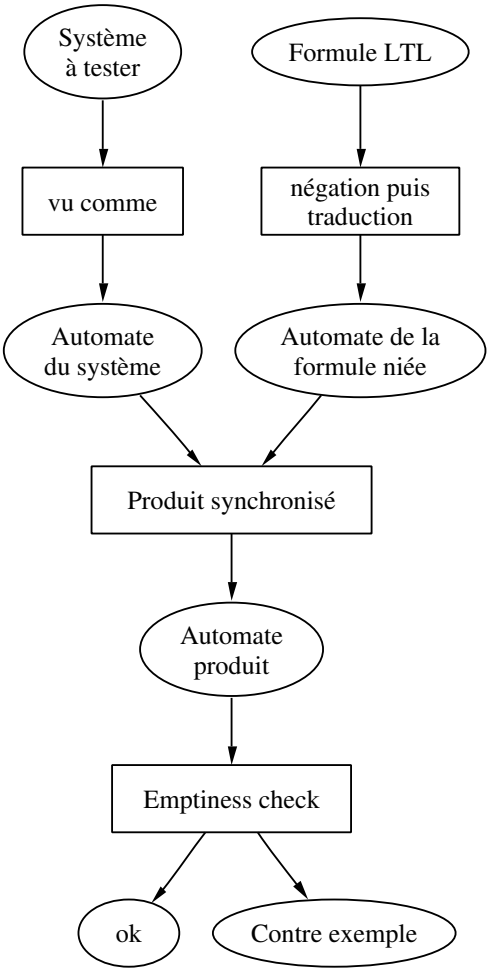


FIG. 2.5 – L'approche par automate du *model checking*.

6. Dans un automate de Rabin, l'ensemble d'acceptation est défini par : $\mathcal{F} = \{(\mathcal{E}_1, \mathcal{F}_1), ..., (\mathcal{E}_n, \mathcal{F}_n)\}$ où $\mathcal{E}_i, \mathcal{F}_i \subseteq \mathcal{Q}$. Une exécution est acceptée s'il existe un i tel qu'elle visite un nombre fini de fois des états de $\mathcal{E}_i$ *et* un nombre infini de fois des états de $\mathcal{F}_i$.
7. L'ensemble d'acceptation d'un automate de Streett est défini de la même manière que celui d'un automate de Rabin, mais la condition est différente. Une exécution est acceptée si, pour tout i, elle visite un nombre fini de fois des états de $\mathcal{E}_i$ *ou* un nombre infini de fois des états de $\mathcal{F}_i$.

Les *automates alternants* permettent de décrire de manière compacte les modèles d'une formule de LTL [52, 53].

8. Un automate alternant est un quadruplet $\mathcal{A}l(\mathcal{Q}, \mathcal{I}, \delta, \mathcal{F})$ sur l'alphabet Σ avec $\delta : \mathcal{Q} \times \Sigma \rightarrow \mathcal{B}^+(\mathcal{Q})$ où $\mathcal{B}^+(\mathcal{Q})$ est l'ensemble des combinaisons booléennes positives d'éléments de $\mathcal{Q}$.
   Par exemple $\delta(q, a) = \{q_1, q_2\}$ peut être écrit $\delta(q, a) = q_1 \vee q_2$. Dans un automate alternant, $\delta(q, a)$ peut être une formule arbitraire de $\mathcal{B}^+(\mathcal{Q})$. Par exemple, $\delta(q, a) = (q_1 \wedge q_2) \vee q_3$ montre que le mot $a\omega$ est accepté lorsque l'on est dans l'état q si ω est accepté depuis q_1 et q_2 ou depuis q_3.
   Une séquence de $\mathcal{A}l$ sur un ω-mot w est un arbre (fini ou infini) étiqueté par $\mathcal{Q}$ tel que : $\tau(\epsilon) \in \mathcal{I}$ et si p est une position de longueur i telle que $\tau(p) = q$ et $\omega(i) = a$ et $\delta(q, a) = \Phi$, l'ensemble $\{p.1, ..., p.n\}$ est tel que $\tau(p.1) \wedge ... \wedge \tau(p.n) \models \Phi$.

Muller et al. [52] introduisent ensuite les automates alternants *faibles* (notés WAA) avec l'ensemble $\mathcal{F}$ d'acceptation qui vérifie les deux conditions suivantes : $\mathcal{F} \subseteq \mathcal{Q}$ et $\mathcal{Q}$ est partitionné en $\mathcal{Q}_i$ tel que $\mathcal{Q}_i \subseteq \mathcal{F}$ ou $\mathcal{F} \cap \mathcal{Q}_i = \emptyset$. Cela définit un ordre partiel sur les $\mathcal{Q}_i$. Rohde introduit dans sa thèse les automates alternants très faibles, leurs conditions d'acceptation résident uniquement dans l'existence d'un ordre partiel sur $\mathcal{Q}$ tel que $\forall q \in \mathcal{Q}$ tous les états de $\delta(q)$ sont plus petits ou égaux à q. Kupferman et Vardi [43, 44], Vardi [69] ont étudié leurs propriétés ainsi que leur utilité dans le *model checking*.

Les automates alternants seront utilisés section 2.3.1.7 pour traduire une formule de façon simple.

Un automate de Büchi étiqueté sur les états (figure 2.6(a)) peut être transformé en un automate de Büchi étiqueté sur les transitions soit en *poussant* les étiquettes d'un état sur toutes ses transitions sortantes (figure 2.6(b)), soit en les *poussant* sur les transitions entrantes (figure 2.6(c)). Dans ce dernier cas, il est nécessaire de rajouter un état initial avec des transitions vers les anciens états initiaux.

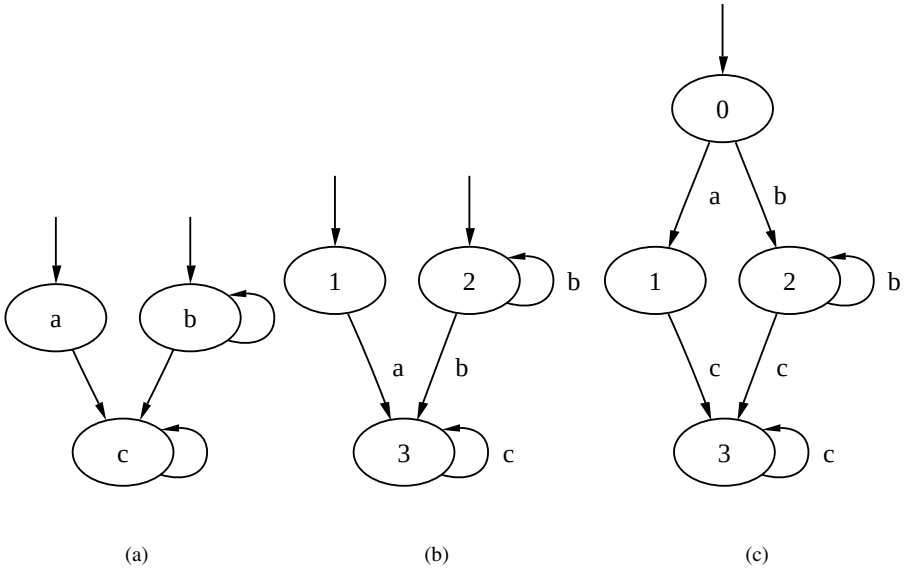


FIG. 2.6 – Transformation d'un automate étiqueté sur les états en un automate étiqueté sur les transitions.

Un automate de Büchi généralisé peut être transformé en un automate de Büchi classique. Cette transformation est appelée dégénéralisation Clarke et al. [15, section 9.2.2].

Considérons un automate de Büchi généralisé étiqueté sur les états $\mathcal{G} = \langle AP, \mathcal{Q}, \delta, l, \mathcal{I}, \mathcal{F} \rangle$.

Pour construire l'automate dégénéralisé, les états de l'automate sont dupliqués $r = |\mathcal{F}|$ fois, et dans la i^{e} copie on s'arrange pour sauter dans la copie suivante à partir de chaque état de l'ensemble d'acceptation $\mathcal{F}_i$. Si $|\mathcal{G}| = n$, on obtient $(r + 1)n$ états. Plus formellement, on définit l'automate dégénéralisé $\langle AP, \mathcal{Q}', \delta', l', \mathcal{I}', \mathcal{F}' \rangle$ par

- $\mathcal{Q}' = \mathcal{Q} \times \{0, \ldots, r\}$
- $\mathcal{I}' = \mathcal{I} \times \{0\}$
- $\mathcal{F}' = \mathcal{Q} \times \{r\}$
- $l'((q,j)) = l(q)$
- $\delta'((q,j)) = \{(q',j') \mid q' \in \delta(q) \text{ et } j' = acc_j(q)\}$ avec $acc_j(q) = \begin{cases} 0 \text{ si } j = r \\ j+1 \text{ si } q \in \mathcal{F}_j \\ j \text{ sinon} \end{cases}$

Cette construction s'adapte facilement au cas des automates étiquetés sur les transitions.

2.3 Traduction de formules LTL en ω-automates

Dans l'approche par automate (section 2.1.8) du *model checking*, une formule φ est traduite en un automate A_φ. Cet automate est ensuite synchronisé avec l'automate représentant le système. La taille de ce produit synchronisé est linéaire par rapport à la taille de l'automate système et linéaire par rapport à la taille du tableau.

Dans le pire cas, l'automate a une taille de l'ordre de $O(2^n)$, où n est la taille de la formule φ. Le *model checking* d'un système de taille m pour une formule de longueur n a donc une complexité en pire cas de l'ordre de $O(m.2^n)$.

Cette complexité motive la recherche d'algorithmes de traduction de formule construisant des automates efficaces, c'est-à-dire avec peu d'états, peu de transitions, et des conditions d'acceptation simples à vérifier.

Nous détaillons ces algorithmes dans la section 2.3.1. De tels algorithmes sont éventuellement précédés d'un travail d'optimisation sur la formule (section 2.3.2). De même, il existe des techniques pour optimiser les automates produits (section 2.3.3).

2.3.1 Algorithmes de traduction de formules LTL en ω-automates

La figure 2.7 présente une généalogie (très incomplète) des méthodes par tableau de traductions de formules LTL en ω-automates. Elle permet de situer les algorithmes présentés ici les uns par rapport aux autres. L'algorithme LTL2BA, présenté section 2.3.1.7, n'apparaît pas car ce n'est pas une construction par tableau. Les nœuds jaunes indiquent les algorithmes produisant des ω-automates étiquetés sur les états, tandis que les roses marquent ceux qui produisent des automates étiquetés sur les transitions. Nous discuterons de la différence section 2.3.1.5.

2.3.1.1 Première traduction : approche combinatoire

La première traduction d'une formule φ vers un automate de Büchi fut proposée par Wolper et al. [79], dans un cadre non spécifique à LTL. Cette traduction produit un automate dont le nombre d'états est systématiquement de l'ordre de $2^{O(|\varphi|)}$.

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.

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 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 compatibles. 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 étiquetés par l'ensemble des littéraux apparaissant dans l'état source. (L'automate ainsi produit est équivalent à un automate étiqueté sur les places, 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 s'assurer que les sous-formules du type $a \,\mathsf{U}\, b$ n'acceptent pas des séquences vérifiant a infiniment sans jamais vérifier b. Pour cela, il suffit d'imposer que tout chemin passe

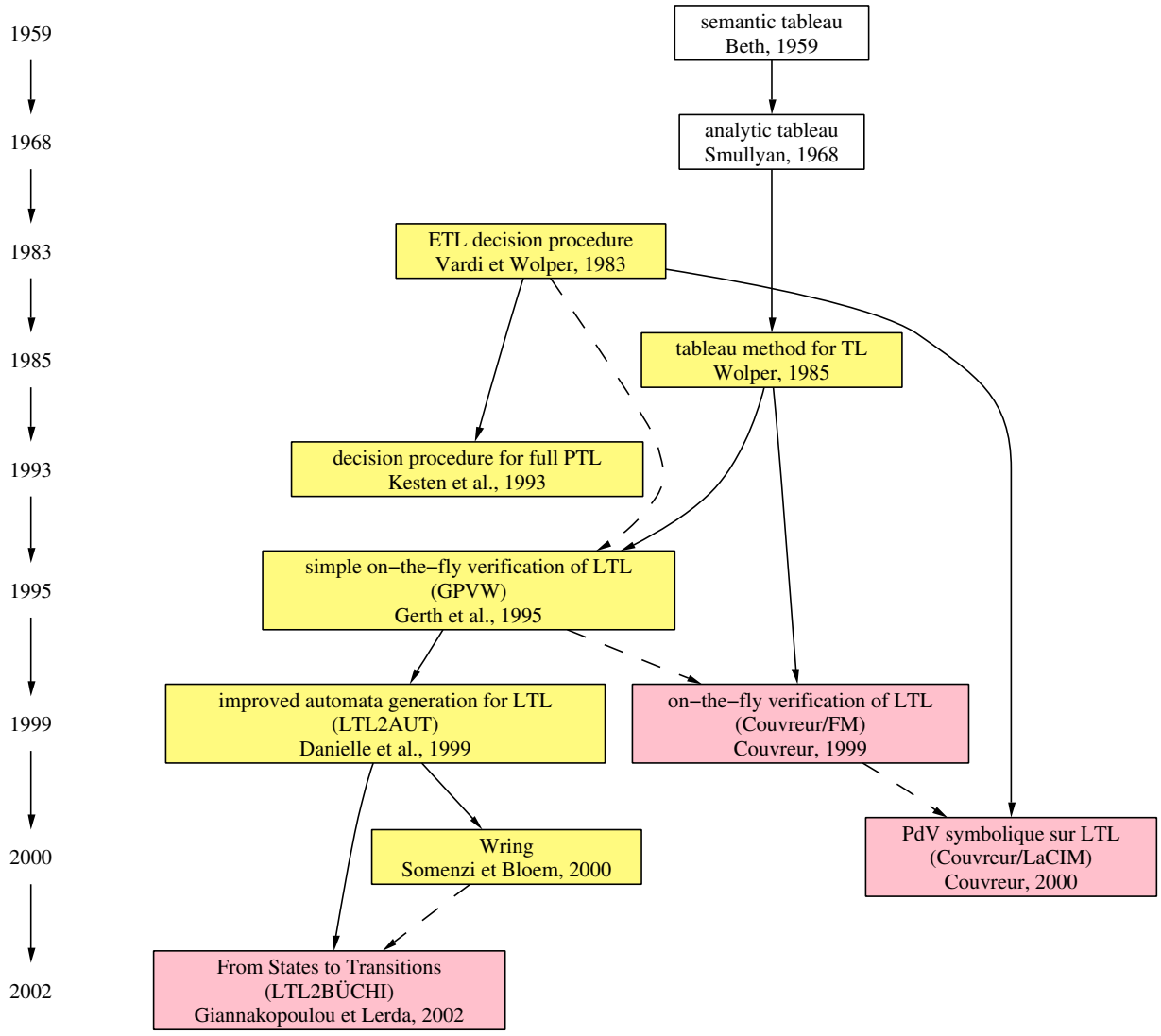


FIG. 2.7 – Généalogie des méthodes par tableau pour la traduction de formules LTL en ω-automates.

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 :

$$\mathcal{F} = \bigcup_{(a \mathsf{U} b) \in cl(\varphi)} \{\mathcal{F}_{a \mathsf{U} b}\} \quad \text{où} \quad \mathcal{F}_{a \mathsf{U} b} = \{s \in S \mid b \in s \vee (a \mathsf{U} b) \notin s\} \tag{2.2}$$

Wolper et al. [79], qui considèrent des automates de Büchi avec un seul ensemble d'états d'acceptation, exprime ces multiples conditions d'acceptions sous la forme d'un second automate, synchronisé par la suite avec celui de la formule. Wolper [77] présente cet algorithme dans un cadre restreint à LTL, et en construisant un automate de Büchi généralisé.

Couvreur [21] propose une traduction similaire tirant partie des BDDs. Nous en reparlerons sections 2.3.1.9.

2.3.1.2 Seconde traduction : raffinements successifs

La première construction d'automate dont la taille ne soit pas celle du pire cas est présentée par Kesten et al. [42], Manna et Pnueli [49]. En fait il ne s'agit pas à proprement parler de la construction d'un automate, mais d'un algorithme pour décider si une formule logique est satisfiable ou non. Il se trouve que cet algorithme construit un graphe orienté, étiqueté sur les états, et dont les chemins ne sont acceptés que si les formules du type $a \mathsf{U} b$ ou $\mathsf{F} b$ sont vérifiées. Il peut donc facilement être vu comme la construction d'un automate de Büchi généralisé.

La logique est à peu près la même que dans l'algorithme précédent : on construit un ensemble d'états qui sont étiquetés par des ensembles de formules compatibles de $cl(\varphi)$, mais les états sont *choisis* de façon à couvrir la formule à tester. (Une relation de comparaison entre ces ensembles de formules permet de caractériser la couverture.)

L'automate est construit de façon itérative. La première version de l'automate ne possède que les états initiaux (toutes les sous-conjonctions de φ à vérifier dans le présent) reliés à un futur générique $\top$ acceptant tout et bouclant sur lui même. Cette première version de l'automate ne permet donc que de vérifier les propriétés de l'instant présent.

Les versions suivantes de l'automate sont construites en *corrigeant* les transitions insatisfaisantes. Une transition est insatisfaisante si elle part d'un état qui couvre $a \wedge \mathsf{X} f$ et n'arrive pas dans un état qui couvre f. La correction consiste à créer tous les états couvrant f, et à rediriger les transitions vers cet état. Ces corrections sont effectuées jusqu'à ce qu'il n'y en ait plus à faire.

Pour n'accepter que les exécutions dont les promesses $a \mathsf{U} b$ et $\mathsf{F} b$ sont tenues, il suffit de définir les mêmes conditions d'acceptation que dans l'algorithme précédent (équation 2.2).

Procéder ainsi (par corrections successives) permet aussi de gérer le fragment passé de LTL.

2.3.1.3 GPVW

L'algorithme de traduction proposé par Gerth, Peled, Vardi, et Wolper [29] construit un automate de Büchi généralisé étiqueté sur les états. Il est basé sur la preuve par tableau présentée section 2.1.7. Mais le tableau, comme celui présenté figure 2.4, est construit implicitement à travers une fonction récursive, Expand. Initialement, Expand reçoit un nœud étiqueté par la formule à satisfaire, puis par réécriture elle décompose les formules de ce nœud et l'étend récursivement en suivant le même algorithme que celui illustré figure 2.4 (page 13).

Un état de l'automate est créé lorsqu'Expand arrive sur un nœud cohérent[9] $\frac{D}{F}$ dans lequel F ne contient plus que des propositions atomiques, leur négation, ou des formules préfixées par X. Ces nœuds du tableaux correspondent à la définition d'état donnée page13, ils sont encadrés sur la figure 2.8(a).

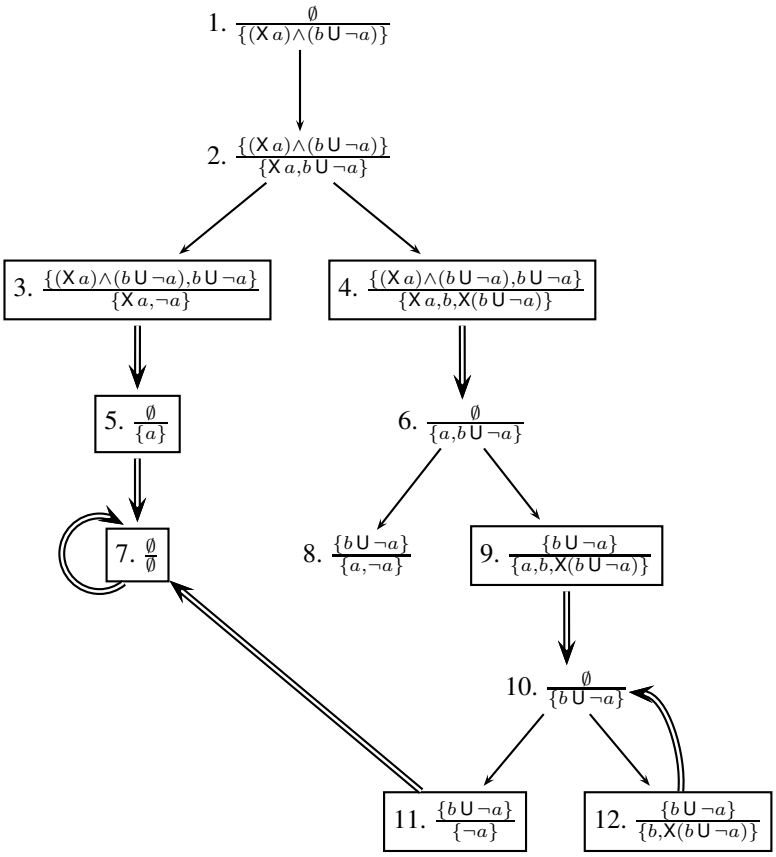
Chaque état de l'automate correspondant à un état $\frac{D}{F}$ du tableau est étiqueté par l'ensemble des valeurs de 2^{AP} compatibles avec les formules de F. Les transitions entre états sont établies de façon à respecter l'ordre induit par le tableau.

Enfin, les conditions d'acceptation (généralisées) sont posées comme dans le premier algorithme (équation 2.2).

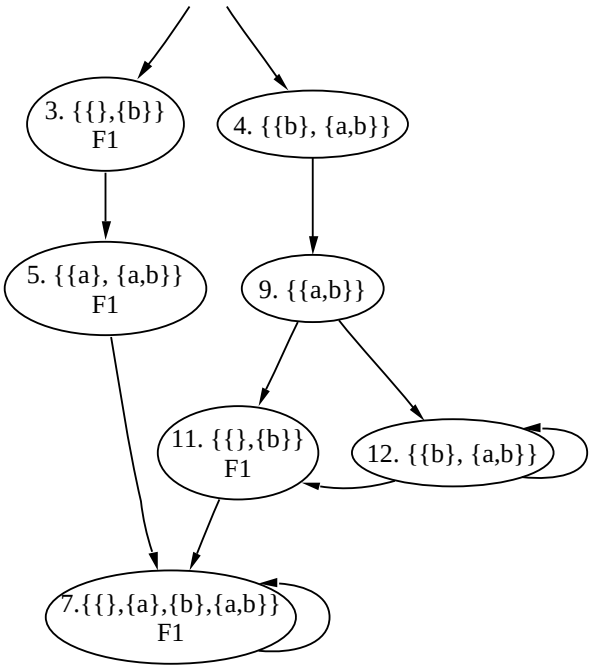
Par exemple l'automate traduisant la formule $(\mathsf{X} a) \wedge (b \mathsf{U} \neg a)$ est représenté figure 2.8(b). Les états portent le nom des nœuds du tableau correspondant (figure 2.8(a)) pour faciliter la comparaison. Dans cet exemple, les conditions de Büchi généralisées $\mathcal{F} = \{\mathcal{F}_1\}$ sont constituées d'un seul ensemble[10], $\mathcal{F}_1$, contenant les états tenant les promesses de la sous-formule $b \mathsf{U} \neg a$. Ces états sont étiquetés par F_1 sur la figure 2.4.

Intuitivement, il est facile de comprendre pourquoi cet algorithme fonctionne. Une preuve par tableau (section 2.1.7) montre la satisfaisabilité 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.

[9] C'est-à-dire ne contenant pas à la fois p et $\neg p$, et ne contenant pas $\bot$.
[10] L'automate produit pour cet exemple particulier peut donc être vu comme un automate de Büchi classique



(a)



(b)

FIG. 2.8 – Tableau et automate de Büchi généralisé étiqueté sur les états pour la formule $(\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$, traduite selon GPVW.

– 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 2.8(a), et se retrouvent donc dans l'automate de la figure 2.8(b). Le caractère *fini* du nombre d'occurrences consécutives de b dans le préfixe du dernier type de séquences est imposé par les conditions d'acceptation de l'automate.

Un point mis en avant par les auteurs 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. Les états initiaux sont construits à partir de φ (états 3 et 4 dans notre exemple). Puis, lorsqu'une branche doit être d'explorée plus en avant (par exemple les étiquettes de 3 sont satisfaites et on souhaite connaître les successeurs possibles) l'algorithme est appliqué localement sur les formules en X de l'état courant (dans le nœud 3 du tableau, $\mathsf{X}\, a$ indique que les nœuds suivant doivent vérifier a, donc l'algorithme est appliqué sur a pour trouver les nœuds correspondants à l'instant suivant). 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*. Les auteurs proposent ensuite de transformer cet automate en un automate de Büchi classique (un seul ensemble d'états d'acceptation). Cette transformation (section 2.2) multiple le nombre d'états de l'automate par le nombre d'ensembles d'acceptation (c'est-à-dire, ici, par le nombre de sous-formules du type $a\,\mathsf{U}\,b$ ou $\mathsf{F}\,b$). Cette dégénéralisation peut se faire elle aussi à la volée.

L'automate de Büchi généralisé étiqueté sur les états peut être interprété comme un automate de Büchi étiqueté sur les transitions en poussant les étiquettes sur les transitions entrantes et en ajoutant un nouvel état initial. La figure 2.9 montre l'automate obtenu pour notre exemple. Une implémentation de cet algorithme, produisant un automate étiqueté sur les transitions comme celui de la figure 2.9 est proposée par Rönkkö [57].

2.3.1.4 LTL2AUT

LTL2AUT [22, 23] est un raffinement de GPVW. Le cœur de l'algorithme (c'est-à-dire la génération d'un automate à partir d'une construction de tableau) est le même, mais quelques points précis ont été améliorés. Nous trouvons plus facile de discuter ces améliorations sur le tableau que sur l'algorithme.

La raison pour laquelle un nœud du tableau, $\frac{D}{F}$, doit conserver dans D l'ensemble des formules déjà récrites, est que certains opérateurs (U et F) font des promesses que l'on cherchera par la suite à exprimer avec des conditions d'acceptation.

Les auteurs de GPVW notaient déjà qu'il était possible d'oublier certaines sous-formules après leur réécriture. Par exemple, $\frac{\emptyset}{\{a \wedge b\}}$ peut-être récrit en $\frac{\emptyset}{\{a,b\}}$ au lieu de $\frac{\{a \wedge b\}}{\{a,b\}}$. Seules les sous-formules de type F ou U, ou celles qui apparaissent en argument de F ou en second argument de U doivent être conservées, car elles sont nécessaires à l'établissement des conditions d'acceptation.

L'intérêt de stocker moins de formules dans D est d'augmenter les chances de fusion des nœuds du tableau (rappelons que deux états $\frac{D}{F}$ et $\frac{D'}{F'}$ ne sont fusionnés que si $D = D'$ et $F = F'$), et donc de diminuer le nombre d'états de l'automate.

En réalité il est possible de fusionner beaucoup plus de nœuds lorsqu'on réalise ce que cette fusion signifie. Deux états peuvent être fusionnés s'ils imposent les mêmes conditions sur l'instant présent et sur le futur. Dans un état $\frac{D}{F}$ du tableau, les conditions portant sur l'instant présent sont les littéraux apparaissant dans F et les conditions portant sur le futur sont les formules préfixées par X apparaissant dans F. En d'autres termes, deux états $\frac{D}{F}$ et $\frac{D'}{F'}$ peuvent être fusionnés si $F = F'$. Il faut juste prendre garde à réunir D et D' (i.e., la fusion de $\frac{D}{F}$ et $\frac{D'}{F}$ est $\frac{D \cup D'}{F}$) pour ne pas perdre les formules de type $a\,\mathsf{U}\,b$, nécessaires à l'établissement des conditions d'acceptation. C'est ce que proposent les auteurs de LTL2AUT, sous une forme un peu plus compliquée.

Par exemple la figure 2.10 montre le tableau construit implicitement par GPVW lors de la traduction de $(a\,\mathsf{U}\,b) \vee (c\,\mathsf{U}\,b)$: les états 8, et 2 peuvent être fusionnés, tout comme 10 et 3, ainsi que 5 et 6. Cette fusion peut se faire directement lors de la construction du tableau. LTL2AUT construit le tableau de la figure 2.11. Le terme $c\,\mathsf{U}\,b$, dans le nœud 5, a été ajouté lors de la fusion du nœud 5 et du nœud 6.

Les automates correspondants à chacun de ces tableaux sont donnés figure 2.12. Plutôt que d'étiqueter chaque état de l'automate par un ensemble d'ensembles de propositions atomiques compatibles avec l'état du tableau, nous avons simplement étiqueté l'automate avec les propositions du tableau (c'est-à-dire a pour $\{\{a\}, \{a, b\}\}$). Comme la for-

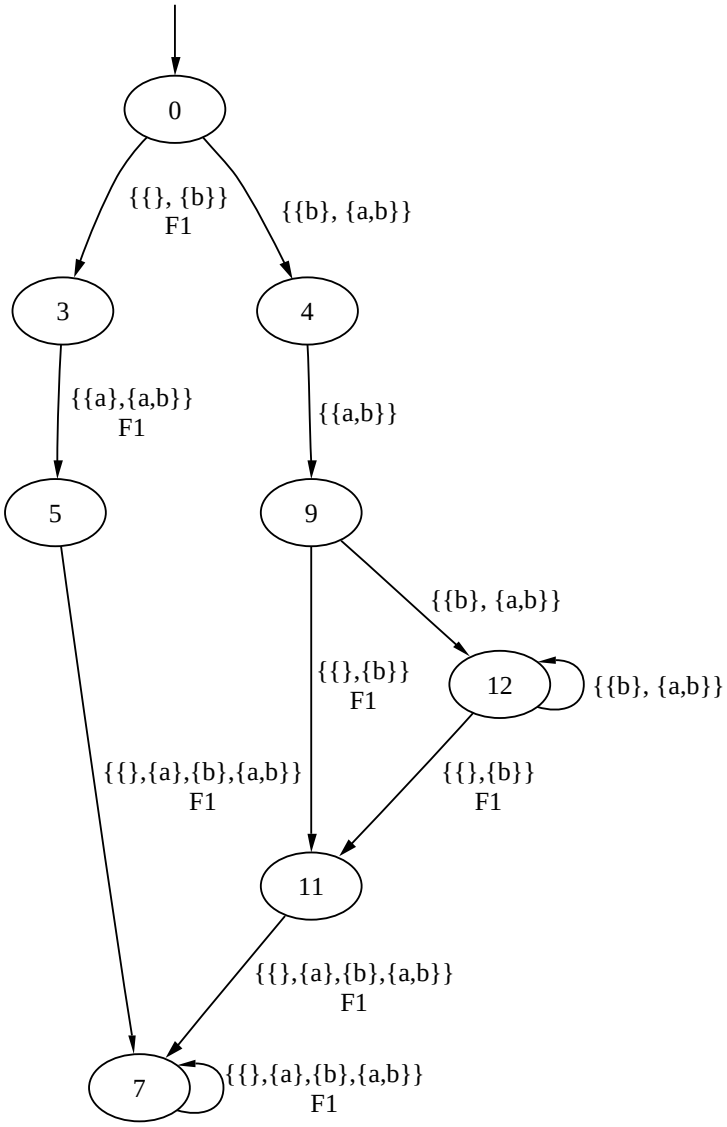


FIG. 2.9 – Automate de Büchi généralisé pour la formule $(\mathsf{X}\, a) \wedge (b\, \mathsf{U} \neg a)$ généré selon la méthode GPVW, mais étiqueté sur les transitions. À comparer avec la figure 2.8(b).

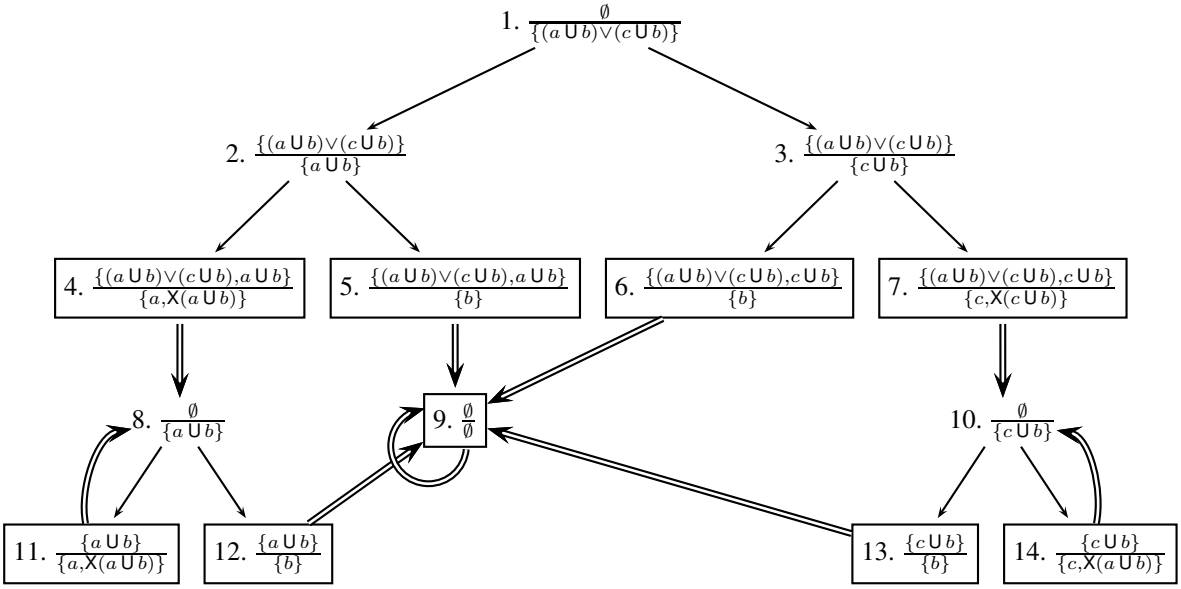


FIG. 2.10 – Traduction de $(a\,\mathsf{U}\,b)\vee(c\,\mathsf{U}\,b)$ avec l'algorithme GPVW.

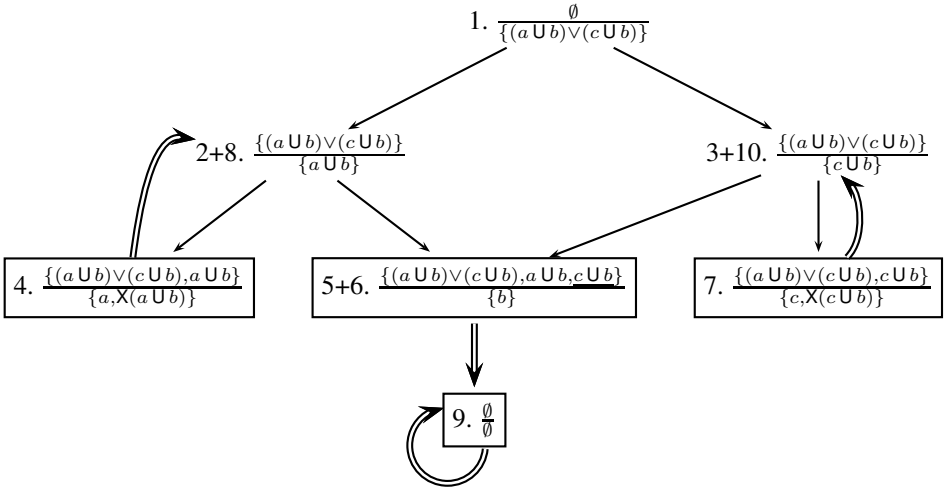


FIG. 2.11 – Traduction de $(a\,\mathsf{U}\,b)\vee(c\,\mathsf{U}\,b)$ avec l'algorithme LTL2AUT.

mule contient deux sous-formules en U, les conditions d'acceptation des automates produits sont constituées de deux ensembles : F_1 est l'ensemble des états où soit $a \mathsf{U} b$ n'apparaît pas, soit b apparaît, de même F_2 est l'ensemble des états où soit $c \mathsf{U} b$ n'apparaît pas, soit b apparaît. Les états sont étiquetés par les noms des ensembles d'acceptation auxquels ils appartiennent.

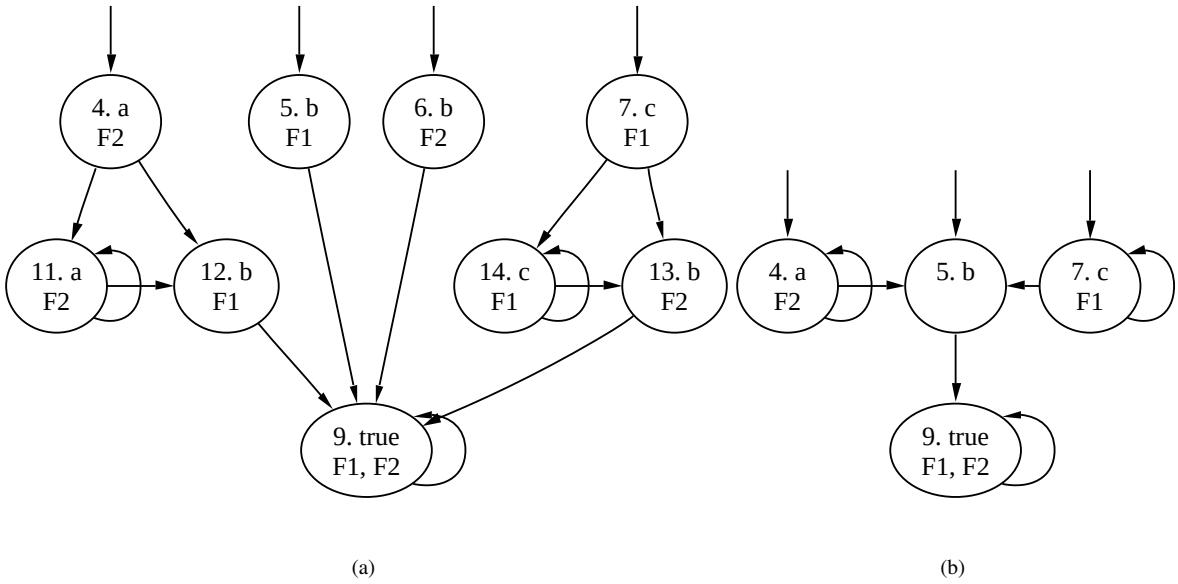


FIG. 2.12 – Automates de Büchi généralisé étiqueté sur les états pour la formule $(a \mathsf{U} b) \vee (c \mathsf{U} b)$, traduits selon (a) GPVW, (b) LTL2AUT. À comparer avec les tableaux figure 2.10 et 2.11 respectivement.

Si la méthode LTL2AUT a un fort impact sur la formule $(a \mathsf{U} b) \vee (c \mathsf{U} b)$, il faut noter qu'elle n'a aucun effet sur $(\mathsf{X} a) \wedge (a \mathsf{U} b)$: le tableau et l'automate de la figure 2.8 sont ceux construits par GPVW aussi bien que LTL2AUT, car aucune des deux branches du tableau ne possède d'états partageant les mêmes présent et futur.

2.3.1.5 Couvreur/FM

Avant de présenter l'algorithme de Couvreur [20], que nous appellerons Couvreur/FM, il convient d'insister sur la différence entre GPVW et LTL2AUT. L'algorithme LTL2AUT associe un état $\frac{D}{F}$ du tableau à un état de l'automate. F contient des formules à vérifier à l'instant présent (elles serviront à étiqueter l'état) et des formules à vérifier dans le futur (elles seront vérifiées par les successeurs de l'état présent). La force de LTL2AUT par rapport à GPVW, est que tous les états du tableau qui possèdent le même F sont associés au même état de l'automate. D, en revanche, n'est pas un critère de décision pour la fusion d'états.

Lorsque les étiquettes correspondant aux propriétés à vérifier dans l'instant présent sont poussées sur les transitions entrantes, cette réutilisation des états n'est plus optimale. En effet, si l'arc entrant représente les formules de l'instant présent, on peut considérer que l'état sur lequel cet arc arrive symbolise les formules à tester dans le futur. Ainsi, en poussant les formules à tester sur les arcs nous devrions être capables de fusionner des états $\frac{D}{F}$ et $\frac{D'}{F'}$ dès que les formules préfixées par X sont identiques dans F et F' (c'est-à-dire si les états partagent le même futur) indépendamment des autres littéraux apparaissant dans F ou F'.

Cela se voit facilement sur la figure 2.9 (page 22). Les états 5, 7, et 11 partagent le même futur. Cela se remarque sur l'automate de la figure 2.9 parce que tous les chemins empruntables à partir de ces états sont étiquetés identiquement. Cela se constate aussi sur le tableau de la figure 2.8(a) (page 20) car les ensembles de formules préfixées par X sont identiques pour chacun de ces états (en effet, il n'y en a aucune). De même, les états 9 et 12 partagent le même futur : $\mathsf{X}(b \mathsf{U} \neg a)$.

La raison pour laquelle ces états ne peuvent pas être fusionnés dans les algorithmes précédents est que les conditions d'acceptation sont posées sur les états, mais portent sur les formules du présent qui ont été poussées sur les arcs. L'arc et sa destination sont donc nécessairement liés du point de vue des conditions d'acceptation.

La construction de Couvreur [20], résout ce problème en faisant porter les conditions d'acceptation sur les arcs et non sur les états. Ceci permet de bien séparer le présent (l'arc) du futur (la destination de l'arc), et ainsi de partager les futurs.

Du point de vue du tableau, l'approche consiste à considérer chaque strate comme un futur. Dans l'état initial, le futur correspond à la formule à satisfaire (par exemple $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$), c'est-à-dire à la racine du tableau. Pour que ce futur soit satisfait, il faut que soit $\neg a$, soit b, soient vérifiées dans le présent. Ces sous-formules apparaissent dans les feuilles de cette strate du tableau. Chacune de ces sous-formules peut mener à un futur différent.

En d'autres termes, les états de l'automate correspondent maintenant aux pré-états du tableau, et les transitions de l'automate traduisent les états du tableau. La figure 2.13 montre le tableau et l'automate obtenus pour $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$ avec cette méthode. Les doubles flèches du tableau sont étiquetés par les sous-formules à vérifier, et les promesses à tenir (formules du type $a \,\mathsf{U}\, b$ ou $\mathsf{F}\, b$). Pour chaque promesse, on définit un ensemble d'acceptation contenant toutes les transitions qui ne font pas cette promesse. Dans l'exemple de la figure 2.13(b), l'ensemble F_1 contient toutes les transitions qui ne promettent pas $b \,\mathsf{U}\, \neg a$ ou qui contiennent $\neg a$[11].

L'automate construit est un automate de Büchi généralisé étiqueté sur les transitions. Une séquence de l'automate est acceptée si elle passe infiniment souvent par au moins une transition de chaque ensemble d'acceptation.

Haddad et Vernadat [38] montrent ce même algorithme, mais en construisant un automate — qu'ils ont baptisé automate à promesses — dont les ensembles de transitions d'acceptation sont définis de façon symétrique à Couvreur/FM. C'est-à-dire que si $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$ est l'ensemble des ensembles d'acceptation de l'automate produit par Couvreur [20], alors $P = \{\mathcal{F} \setminus \mathcal{F}_1, \mathcal{F} \setminus \mathcal{F}_2, \ldots, \mathcal{F} \setminus \mathcal{F}_n\}$ est l'ensemble des ensembles de promesses à tenir. La condition d'acceptation d'un automate à promesses est symétrique à celle d'un automate de Büchi : une séquence est acceptée si pour chaque ensemble de promesse $\mathcal{P}_i$, elle passe infiniment souvent par une transition $t \notin \mathcal{P}_i$. L'automate à promesses pour l'exemple $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$ est donné figure 2.13(c).

Ces deux types d'automate se manipulent de façon similaire. Un algorithme (dû à Couvreur) est présenté section 2.4.2.

Un autre aspect intéressant de cette traduction, est la représentation de la relation de transition de l'automate sous la forme d'un BDD (cf. section 2.6.2). Ceci permet au BDD de fusionner ou de simplifier certains arcs automatiquement (figure 2.14). Ce point sera aussi discuté section 2.3.1.6.

2.3.1.6 Wring

Bloem [6], Somenzi et Bloem [60] proposent une traduction de formules en trois étapes.

1. simplification de la formule avec des règles de réécritures (ce point sera discuté section 2.3.2)
2. une traduction proprement dite
3. une simplification par simulation de l'automate produit (section 2.3.3.1)

La traduction repose sur un calcul de couverture optimisée. Toute formule de logique propositionnelle peut être mise sous forme normale disjonctive : une somme de produits. Sans imposer de conditions supplémentaires, la forme obtenue n'est pas unique. Par exemple $(a \wedge \neg b \wedge \neg c) \vee (b \wedge c) \vee (\neg b \wedge c) \vee (a \wedge b \wedge \neg c)$ et $a \vee c$ sont deux formes normales disjonctives équivalentes. La réduction d'une forme normale disjonctive est un problème classique de logique, résolu par des techniques telles que les tableaux de Karnaugh ou encore l'algorithme de Quine-McCluskey.

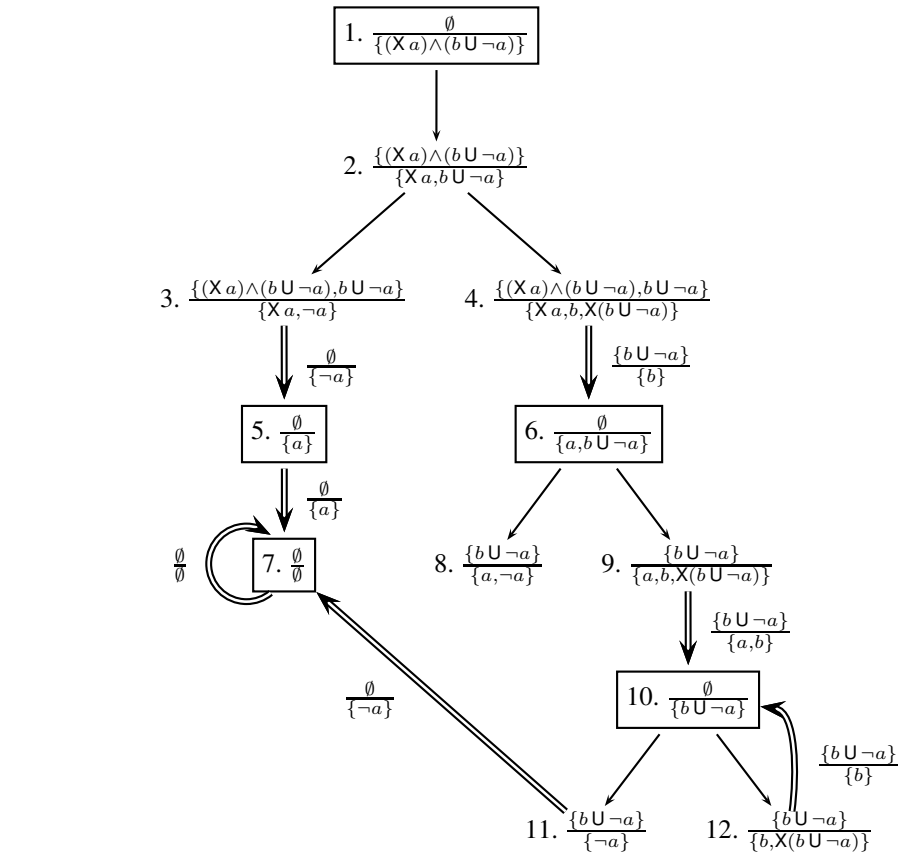
La construction d'un automate pour une formule de logique temporelle par une méthode de tableau crée une sorte de forme normale disjonctive. Par exemple, les nœuds 3 et 4 du tableau de la figure 2.4 (page 13) correspondent à la mise sous forme disjonctive de la formule du nœud 1.

$$(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a) = (\neg a \wedge \mathsf{X}\, a) \vee (b \wedge (\mathsf{X}\, a) \wedge \mathsf{X}(b \,\mathsf{U}\, \neg a))$$

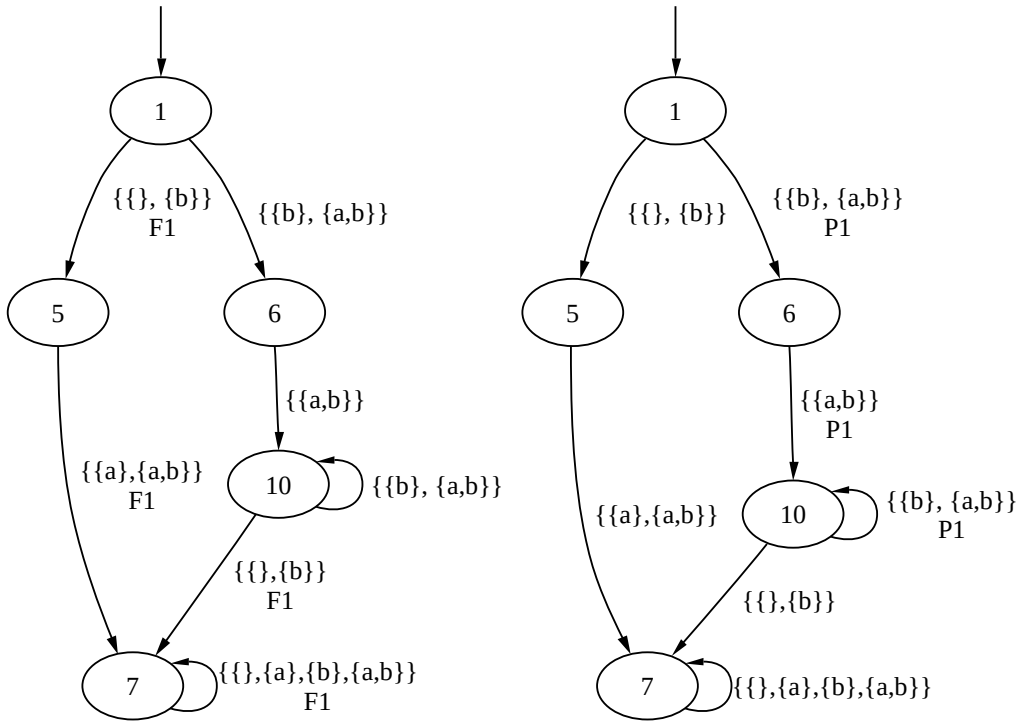
Chaque terme de la disjonction implique la formule complète ; ils sont à juste titre appelés impliquants. L'ensemble de ces impliquants forme la couverture de la formule à tester. Dans la construction d'un automate par une méthode de tableau, ces impliquants deviendront les états de l'automate. Il serait donc intéressant de minimiser leur nombre (trouver une couverture optimale) de la même façon qu'on peut le faire en logique propositionnelle, afin de minimiser le nombre de branches du tableau.

Malheureusement, les algorithmes connus en logique propositionnelle ne sont pas valables dans le cas présent car ils ignorent les promesses faites par les opérateurs LTL. Ainsi, si deux nœuds d'un tableau de logique propositionnelle, $\{a, b\}$ et $\{a, \neg b\}$ peuvent être simplifiés en $\{a\}$, deux nœuds d'un tableau de LTL $\frac{\{a \,\mathsf{U}\, d\}}{\{a, b\}}$ et $\frac{\{a \,\mathsf{U}\, e\}}{\{a, \neg b\}}$ ne peuvent être simplifiés car ils tiennent des promesses différentes.

[11] En réalité, Couvreur/FM définit l'ensemble F_1 comme l'ensemble des transitions qui ne contiennent pas $b \,\mathsf{U}\, \neg a$, sans inclure celles qui contiennent $\neg a$. Dans l'exemple figure 2.13(b), cela supprime la transition $10 \rightarrow 7$ de l'ensemble F_1. Il serait intéressant de justifier cette simplification. Visiblement l'algorithme suppose que des formules telles que $y \wedge (x \,\mathsf{U}\, y)$ ont été simplifiées en y. C'est automatiquement le cas si la relation de transition de l'automate construit est représentée par un BDD. (Intuitivement, la simplification vient du fait que $y \wedge (x \,\mathsf{U}\, y)$ est en fait représenté par $y \wedge (y \vee (x \wedge \mathsf{X}(x \,\mathsf{U}\, y)))$, qui se simplifie directement en y.)



(a)



(b) (c)

FIG. 2.13 – (a) tableau et (b) automate de Büchi pour la formule $(\mathsf{X}\, a) \wedge (b\, \mathsf{U}\, \neg a)$, construits avec l'algorithme de Couvreur [20]. À comparer avec ceux des figures 2.8(a) (page 20) et 2.9 (page 22). (c) automate à promesses pour cette formule.

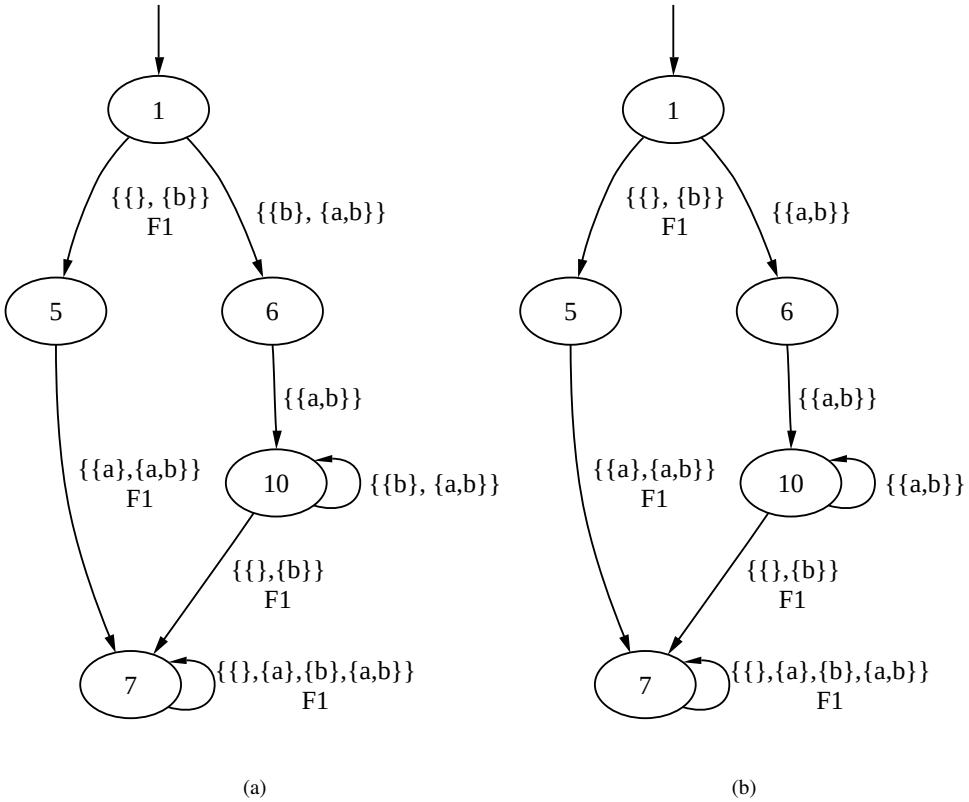


FIG. 2.14 – Simplification effectuée lors d'un encodage par BDD : (a) reprise de la figure 2.13(b), et (b) encodage par BDD. Des étiquettes superflues ont été retirées des arcs $1 \rightarrow 6$ et $10 \rightarrow 10$.

La solution de Couvreur [20] à ce problème est de représenter les promesses sous forme de variables propositionnelles, $a_{a\,\mathsf{U}\,d}$ et $a_{a\,\mathsf{U}\,e}$, dans les formules. Cela permet d'appliquer n'importe quelle technique de minimisation de formules propositionnelles. En l'occurrence Couvreur/FM encode la relation de transition par un BDD : une minimisation (pas forcément optimale) est effectuée automatiquement par le BDD (cf. figure 2.14), et des techniques de calcul d'implicants premiers peuvent être mises en place [18].

Dans le cas de Wring, l'automate construit est un automate étiqueté sur les états. Les *minterms* définissent donc des états et non des transitions. Les auteurs de Wring considèrent cette minimisation comme un problème d'optimisation linéaire, qu'ils résolvent avec une heuristique.

La construction d'un automate de Büchi généralisé étiqueté sur les états à partir de cette couverture est sans surprise. L'automate construit est ensuite réduit par une technique de simulation que nous discuterons en section 2.3.3.1.

Cet algorithme produit des automates systématiquement plus petits (au sens large) que ceux de GPVW et LTL2AUT. Il bat parfois celui de Couvreur ce qui semble indiquer que ce dernier pourrait être amélioré en adaptant certains des pré- ou post-traitements appliqués ici.

Le tableau 2.3 synthétise les exemples d'automates donnés par Couvreur [20, 21] et Bloem [6]. Les deux dernières colonnes donnent les tailles des automates généralisés produits par LTL2BA et Couvreur/LaCIM, présentés dans les sections suivantes.

	GPVW			LTL2AUT			Wring			Couvreur FM			LTL2BA			Couvreur LaCIM		
	ét.	tr.	$\lvert\mathcal{F}\rvert$	ét.	tr.	$\lvert\mathcal{F}\rvert$	ét.	tr.	$\lvert\mathcal{F}\rvert$	ét.	tr.	$\lvert\mathcal{F}\rvert$	ét.	tr.	$\lvert\mathcal{F}\rvert$	ét.	tr.	$\lvert\mathcal{F}\rvert$
$p\,\mathsf{U}\,q$	3	4	1	3	4	1	3	4	1	2	3	1	2	3	1	2	-	1
$p\,\mathsf{U}(q\,\mathsf{U}\,s)$	6	10	2	4	7	2	4	7	1	3	6	2	3	6	2	3	-	2
$\neg(p\,\mathsf{U}(q\,\mathsf{U}\,s))$	7	15	0	7	15	0	6	10	0	3	6	0	3	6	0	3	-	0
$\mathsf{G}\,\mathsf{F}\,p \Rightarrow \mathsf{G}\,\mathsf{F}\,q$	9	15	2	5	11	2	4	7	1	4	9	2	3	5	2	7	-	2
$(\mathsf{F}\ p)\,\mathsf{U}(\mathsf{G}\,q)$	8	15	2	6	17	2	3	4	1	4	10	2	4	10	2	3	-	2
$(\mathsf{G}\,p)\,\mathsf{U}\,q$	5	6	1	5	6	1	5	6	1	4	6	1	4	6	1	4	-	1
$\neg(\mathsf{F}\,\mathsf{F}\,p \Leftrightarrow \mathsf{F}\,p)$	12	16	2	1	1	2	0	0	0	2	3	2	0	0	0	1	-	2

TAB. 2.3 – Comparaison des tailles des automates généralisés produits par GPVW, LTL2AUT, Wring, Couvreur/FM, LTL2BA, et Couveur/LACIM.

2.3.1.7 LTL2BA

Kupferman et Vardi [43] terminent en remarquant que les automates alternants faibles peuvent être utilisés de manière efficace dans le *model checking*. L'algorithme de traduction que propose Gastin et Oddoux [28] construit un automate de co-Büchi étiqueté sur les transitions. Il n'est pas basé sur une preuve par tableau.

Dans une première étape, l'algorithme génère un automate de co-Büchi alternant *très faible* (noté VWAA ; voir section 2.2). Le nombre d'états de cet automate est borné par $|\varphi|$, la taille de la formule à traduire. Gastin et Oddoux [28] définissent la fermeture de la formule à traduire comme l'ensemble des sous-formules LTL qui ne sont pas une conjonction ou une disjonction de formules. La fermeture de φ détermine l'ensemble des états $\mathcal{Q}$ de l'automate. La fonction de transition de l'automate est défini par $\delta : \mathcal{Q} \rightarrow 2^{\Sigma' \times \mathcal{Q}'}$ avec $\mathcal{Q}' = 2^{\mathcal{Q}}$ et $\Sigma' = 2^{\Sigma}$. Elle fait intervenir deux opérateurs $\otimes$ et $\bar{\cdot}$ qui permettent respectivement de synchroniser deux transitions et d'obtenir une forme normale disjonctive d'une formule LTL.

– Pour tout $\mathcal{J}_1, \mathcal{J}_2 \in 2^{\Sigma' \times \mathcal{Q}'}$, on a $\mathcal{J}_1 \otimes \mathcal{J}_2 = \{(\alpha_1 \cap \alpha_2, e_1 \wedge e_2) |\ (\alpha_1, e_1) \in \mathcal{J}_1 \text{ et } (\alpha_2, e_2) \in \mathcal{J}_2\}$.
– pour toute formule LTL φ, on a $\overline{\varphi} = \{\varphi\}$ si φ n'est pas une conjonction ou une disjonction. Sinon, $\overline{\psi_1 \vee \psi_2} = \overline{\psi_1} \cup \overline{\psi_2}$ et $\overline{\psi_1 \wedge \psi_2} = \{e_1 \wedge e_2 | e_1 \in \overline{\psi_1} \text{ et } e_2 \in \overline{\psi_2}\}$.

L'ensemble d'états initiaux $\mathcal{I}$ correspond à $\overline{\varphi}$, et l'ensemble des états acceptation $\mathcal{F}$ correspond aux états associés aux sous-formules de la forme $\psi_1\,\mathsf{U}\,\psi_2$. L'ensemble $\mathcal{F}$ définit la condition d'acceptation de co-Büchi (voir section 2.2).

Pour l'exemple $\varphi = (\mathsf{X}\,a) \wedge (b\,\mathsf{U}\,\neg a)$, la première étape de l'algorithme nous donne l'automate figure 2.15(a). Dans cet automate aucun état n'est étiqueté par φ puisque c'est une conjonction de deux sous-formules : $\psi_1 = \mathsf{X}\,a$ et $\psi_2 = b\,\mathsf{U}\,\neg a$. Les *implications* tracées en traits discontinus distinguent les deux branches du VWAA. Une branche de cet automate traite ψ_1 et l'autre ψ_2. Le VWAA obtenu vérifie bien les conditions d'acceptation de co-Büchi. Une séquence acceptée ne contient qu'un nombre fini de nœuds étiquetés par $b\,\mathsf{U}\,\neg a$ quelque soit la branche choisie (dans cet exemple $\mathcal{F}_0 = \{b\,\mathsf{U}\,\neg a\}$).

Dans une second étape, Gastin et Oddoux [28] construisent un automate de Büchi généralisé étiqueté sur les transitions à partir du VWAA obtenu. L'ensemble des transitions est réduit en conservant uniquement les minimaux d'une *relation partielle* $\preceq$ définie sur les transitions possibles. L'ensemble $\mathcal{T}$ d'ensembles de transitions d'acceptation est

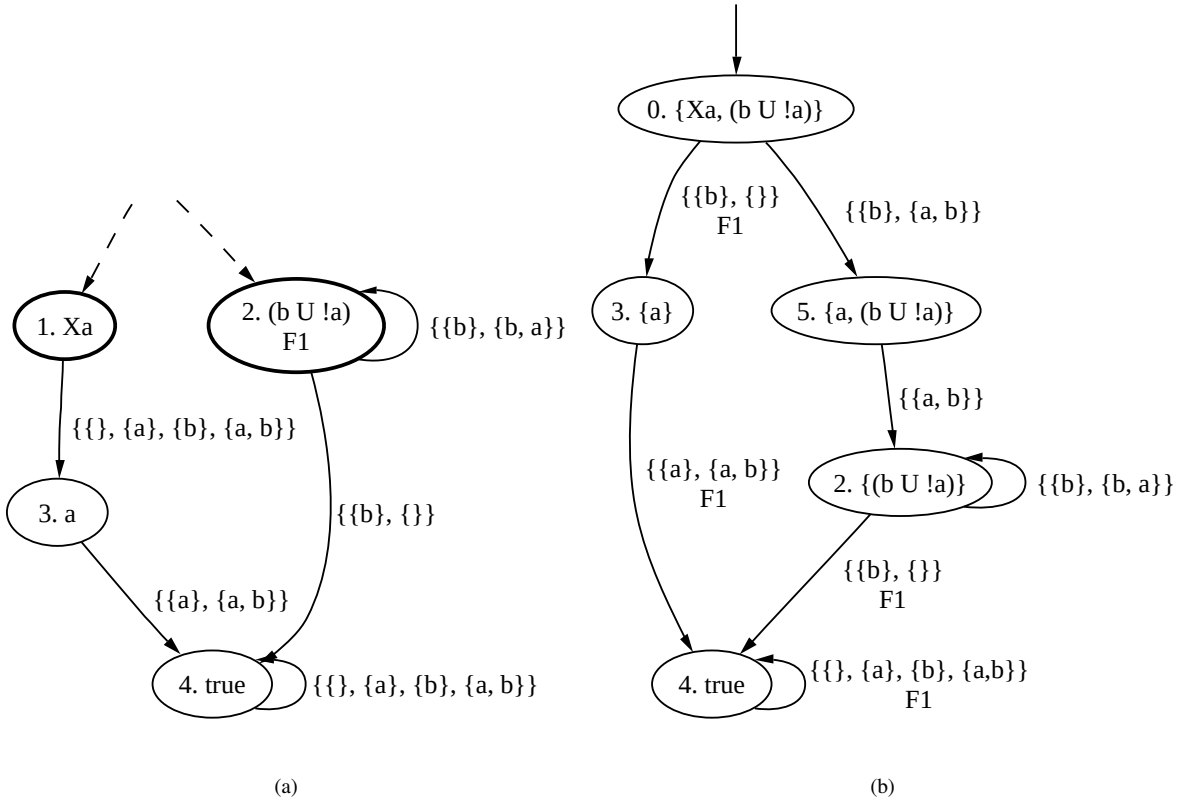


FIG. 2.15 – (a) L'automate alternant très faible obtenu pour la formule $(\mathsf{X}\, a) \wedge (b \,\mathsf{U}\, \neg a)$, et (b) l'automate de Büchi généralisé construit à partir de (a).

défini comme suit. $\mathcal{T} = \{T_f | f \in \mathcal{F}\}$ avec $\mathcal{F}$ l'ensemble d'états d'acceptation du VWAA et $T_f = \{(e, \alpha, e') | f \notin e'$ ou $\exists (\beta, e'') \in \delta(f)$ telles que $\alpha \subseteq \beta$ et $f \notin e'' \subseteq e'\}$.

L'ensemble $\mathcal{T}$ de l'automate 2.15(b) est composé d'un ensemble T_f contenant les quatre transitions étiquetées par F_1. En effet, dans cet exemple $\mathcal{F} = \{b \,\mathsf{U}\, \neg a\}$ et $\{b \,\mathsf{U}\, \neg a\} \subset \delta(b \,\mathsf{U}\, \neg a)$. L'automate généralisé ainsi produit possède au plus $2^{|\mathcal{Q}|}$ états et $|\mathcal{F}|$ ensembles d'acceptation.

Enfin, l'automate généralisé peut être transformé en un automate de Büchi classique avec au plus $n \times 2^n$ états (où n est la taille de la formule). Cette seconde étape n'est nécessaire que pour se comparer à d'autres algorithmes, ou pour utiliser un algorithme de vérification ne supportant pas les conditions d'acceptation de Büchi généralisées.

Des simplifications telles que celles discutées en section 2.3.3 sont effectuées sur chaque automate intermédiaire.

LTL2BA a été implémenté en C en utilisant des morceaux de Spin (section 2.8) comme base de travail, et est comparé avec Spin après avoir appliqué les réécritures de Wring. Le tableau 2.4 effectue la comparaison. Pour les formules (2.3) et (2.4) de la section 2.8 (page 41), Gastin et Oddoux [28] montre que Wring ne construit pas l'automate pour ϕ_8 et ψ_8 en temps et espace raisonnable tandis que LTL2BA produit des automates pour ϕ_8 et ψ_{10}.

	Spin + réécritures		LTL2BA	
	moy.	max.	moy.	max.
temps d'exécution (sec.)	14.23	4521.65	0.01	0.04
états	5.74	56	4.51	39
transitions	14.73	223	9.67	112

TAB. 2.4 – Comparaison entre Spin amélioré et LTL2BA sur 200 formules de taille 10 prises au hasard.

2.3.1.8 LTL2BÜCHI

Giannakopoulou et Lerda [30, 31] ont proposé un algorithme de traduction d'une formule LTL en un automate de Büchi classique étiqueté sur les transitions. L'algorithme est une preuve par tableau présentée dans le même formalisme que GPVW, mais décrite comme une version de LTL2AUT produisant des automates de Büchi étiquetés sur les transitions.

La formule est tout d'abord récrite en utilisant les optimisations de Etessami et Holzmann [25] et celles de Somenzi et Bloem [60] (voir section 2.3.2). L'algorithme de traduction semble similaire à celui présenté par Couvreur [20] deux ans plus tôt, mais visiblement développé de façon indépendante. LTL2BÜCHI utilise des vecteurs de booléens dans chaque nœud pour représenter les conditions d'acceptation qui lui sont associées, et introduit des techniques de comparaison de formules basées sur leur arbre de syntaxe abstrait (comme celui de la figure 2.2 page 11). Cela diffère de Couvreur/FM qui fait tout cela avec des BDD.

À la différence de Couvreur/FM, qui utilise directement l'automate avec conditions d'acceptation généralisées produit (section 2.4.2), les auteurs de LTL2BÜCHI ajoutent une phase de dégénéralisation pour obtenir un automate de Büchi étiqueté sur les transitions simples. Cette dégénéralisation est celle de LTL2BA. L'automate ainsi obtenu peut alors être utilisé par des *emptiness checks* classiques adaptés pour travailler sur les transitions.

2.3.1.9 Couvreur/LaCIM

Une version moderne de traduction par énumération de l'ensemble des combinaisons de sous-formules (dans la lignée de celui présenté section 2.3.1.1) est donnée par Couvreur [21].

Tout comme Couvreur/FM, cet algorithme construit un automate de Büchi généralisé étiqueté sur les transitions, puisque de tels automates sont plus compacts que ceux étiquetés sur les états. Mais à la différence de Couvreur/FM, la construction n'est pas une méthode tableau.

Pour chaque sous-formule f de la formule à traduire, un couple des variables BDD Now$[f]$, Next$[f]$ est créé. Chaque état de l'automate correspond à une affectation cohérente de l'ensemble des variables Now$[\cdot]$ (ce point est identique à l'approche de la section 2.3.1.1). Cette cohérence, ainsi que les transitions entre les différents états, sont imposées par une relation de transition écrite comme une formule BDD liant les variables Now$[\cdot]$ (désignant l'état courant), les variables Next$[\cdot]$ (désignant l'état suivant) et les propositions atomiques (désignant les étiquettes des transitions). Cette relation de transition est construite à l'aide de contraintes (entre ces différentes variables) obtenues lors d'un parcours récursif de la formule.

La tableau 2.5 donne les contraintes imposées pour chaque type de sous-formule rencontrée lors du parcours récursif, ainsi que les éventuelles conditions d'acceptation. Dans ce tableau, p désigne une proposition atomique, f et g sont des formules propositionnelles quelconques.

formule	état initial	contrainte	condition d'acceptation
$\neg p$	p		
f	Now$[f]$		
$\mathsf{X}\, f$	Next$[f]$	Now$[f] \Leftrightarrow f$	
$f \,\mathsf{U}\, g$	Now$[f \,\mathsf{U}\, g]$	Now$[f \,\mathsf{U}\, g] \Leftrightarrow (g \vee (f \wedge \text{Next}[f \,\mathsf{U}\, g]))$	$g \wedge \neg\text{Now}[f \,\mathsf{U}\, g]$
$f \,\mathsf{R}\, g$	Now$[f \,\mathsf{R}\, g]$	Now$[f \,\mathsf{R}\, g] \Leftrightarrow (g \wedge (f \vee \text{Next}[f \,\mathsf{U}\, g]))$	

TAB. 2.5 – Éléments de traduction pour l'algorithme de Couvreur [21].

La traduction de l'automate se fait récursivement par réécriture selon ces règles.

Couvreur [21] présente aussi la façon de traduire les opérateurs du passé, ainsi que toute opérateur exprimable sous la forme d'un ω-automate.

L'avantage de cette méthode par rapport à la première (section 2.3.1.1) tient à ce que l'énumération de toutes les combinaisons d'états et de transitions se fait implicitement à l'aide de la relation de transition sous forme de BDD. Le calcul des successeurs d'un état revient à chercher l'ensemble des affections de variables Next$[\cdot]$ qui, avec les variables Now$[\cdot]$ de l'état courant, satisfont la relation de transition.

Couvreur [21] montre que pour de petites formules, les automates produits peuvent être compétitifs avec ceux des méthodes tableau. Par contre il est facile de construire des formules qui font exploser le nombre d'états : chaque ajout d'un X en tête de formule double les états.

Enfin, l'un des avantages de cette traduction est l'utilisation d'une représentation symbolique (des BDD) de l'automate généré. Cette représentation ouvre la porte aux optimisations symboliques. Par exemple avec des techniques de point fixe il est possible de supprimer de l'automate des états qui ne peuvent appartenir à aucun chemin acceptant. C'est d'ailleurs grâce à cette optimisation que cette méthode produit un automate à un seul état (l'état initial, sans successeur) pour la formule $\neg(\mathsf{F}\,\mathsf{F}\, p \Leftrightarrow \mathsf{F}\, p)$, ainsi que l'indique le tableau 2.3.

La raison pour laquelle le tableau 2.3 n'indique pas le nombre de transitions pour cet algorithme est que la relation de transitions est encodée sous la forme d'un BDD, et non pas explicitement. Lors du produit synchronisé, il est possible de réduire les successeurs d'un état de l'automate de la formule à ceux qui sont compatibles avec un état du système en injectant les propriétés vérifiées par ce dernier dans la relation de transition du premier. Le nombre de transitions

ne peut donc pas être utilisé comme base de comparaison. D'autre part, ce nombre pourrait varier selon la façon de calculer les impliquants premiers,

2.3.2 Pré-traitement : réécriture de la formule

L'optimisation de formules au niveau propositionnel, telle qu'elle fut discutée section 2.3.1.6 (problème d'optimisation, ou utilisation de BDD), ne prend pas en compte les propriétés des opérateurs de logique temporelle. Par exemple au niveau propositionnel, deux formules telles que $\mathsf{F}\,\mathsf{F}\,p$ et $\mathsf{F}\,p$ sont vues comme deux propositions différentes. C'est la raison pour laquelle Couvreur/FM (dont les formules sont implicitement optimisées par des BDD) ne produit pas un automate vide pour la formule $\neg(\mathsf{F}\,\mathsf{F}\,p \Leftrightarrow \mathsf{F}\,p)$. Plusieurs auteurs ont proposé des règles de réécriture, pour simplifier la formule *avant* sa traduction. De telles réécritures ont un coût ridicule par rapport au reste du *model checking* : il s'agit de faire du *pattern matching* sur l'arbre de syntaxe de la formule pour la transformer.

Éventuellement, ce type de réécriture peut être *dirigé* pour prendre en compte des connecteurs traités spécialement lors de la traduction. Par exemple Couvreur [20] traduit $\mathsf{G}\,\mathsf{F}\,\varphi$ de façon spécifique, on aurait donc intérêt à le prendre en compte lors des réécritures.

Somenzi et Bloem [60], Etessami et Holzmann [25] et [6] proposent un système de réécriture de la formule qui permet une réduction portant sur les propriétés des opérateurs de logique temporelle et une réduction au niveau propositionnel. Les systèmes proposé par Somenzi et Bloem d'une part, et Etessami et Holzmann d'autre part ne se recoupent pas totalement.

Etessami et Holzmann [25] mettent en évidence des règles de réécriture pour les langages clos par préfixe ou suffixe. Un langage $\mathcal{L}$ *clos par ajout à gauche* sur Σ^ω et tel que $\forall(\omega, \upsilon) \in \Sigma^\omega \times \Sigma^*, \omega \in \mathcal{L} \Rightarrow \upsilon\omega \in \mathcal{L}$. Un langage $\mathcal{L}$ *clos par suffixe* sur Σ^ω et tel que $\forall\omega \in \mathcal{L}$ si ω' est un suffixe de ω alors $\omega' \in \mathcal{L}$.

Si le langage associé à une formule f (langage composé des mots validant f) est clos par ajout à gauche alors toute formule du type $g\,\mathsf{U}\,f$ (quelque soit g) ou $\mathsf{F}\,f$ peut être récrite en f. S'il est clos par suffixe, alors les formules du types $g\,\mathsf{R}\,f$ ou $\mathsf{G}\,f$ peuvent être récrites en f. Chaque application de l'une de ces règles supprime un opérateur de la formule.

2.3.3 Post-traitements

Cette étape de réduction est appliquée sur la construction de l'automate de Büchi étiqueté sur les transitions. En ce sens, Etessami et Holzmann [25], Somenzi et Bloem [60], Gastin et Oddoux [28] énoncent plusieurs optimisations sur l'automate de la formule. Certaines d'entre elles se retrouvent dans plusieurs algorithmes ou méthodes analysées section 2.3.1.

1. Tout d'abord, il est possible de supprimer les états inaccessibles de l'automate sans changer le langage associé.
2. D'autre part nous pouvons réduire le nombre de transitions et fusionner les étiquettes de plusieurs transitions. Par exemple, si on suppose deux transitions t_1, t_2 de l'automate telles que : $t_1 : (s_0, f, s_1)$ et $t_2 : (s_0, g, s_1)$, on peut alors remplacer t_1 et t_2 par $t : (s_0, f \vee g, s_1)$.
3. Si l'on considère les notions d'*implication* et d'*équivalence* proposées par Gastin et Oddoux [28] et représentées par le tableau 2.6, les deux propositions suivantes sont vérifiées :
   – Si une transition t_1 *implique* une transition t_2 alors on peut supprimer t_2.
   – Si deux états q_1 et q_2 sont *équivalents* alors on peut les identifier.

Automates	$t_1 = (q, \alpha_1, q_1) \Rightarrow t_2 = (q, \alpha_2, q_2)$	$q_1 \Leftrightarrow q_2$
VWAA	$\alpha_2 \subseteq \alpha_1$ et $q_1 = q_2$	$\delta(q_1) = \delta(q_2)$ et $q_1 \in \mathcal{F} \Leftrightarrow q_2 \in \mathcal{F}$
GBA	$\alpha_1 \subseteq \alpha_2$, $q_1 = q_2$ et $\forall T \in \mathcal{T}, t_2 \in T \Rightarrow t_1 \in T$	$\delta(q_1) = \delta(q_2)$ et $\forall(\alpha, q') \in \delta(q_1), \forall T \in \mathcal{T}$, $(q_1, \alpha, q') \in \mathcal{T} \Leftrightarrow (q_2, \alpha, q') \in T$
BA	$\alpha_2 \subseteq \alpha_1$ et $q_1 = q_2$	$\delta(q_1) = \delta(q_2)$ et $q_1 \in \mathcal{F} \Leftrightarrow q_2 \in \mathcal{F}$

TAB. 2.6 – L'*implication* de transition et l'*équivalence* d'états suivant la nature de l'automate (les notations étant celles de la section 2.3.1.7 pour VWAA).

4. Etessami et Holzmann [25] montrent aussi que dans un automate de Büchi, une composante fortement connexe contenant au moins un état d'acceptation peut être réduite à un seul état d'acceptation (appartenant à cette même composante connexe) si l'on ne peut sortir de cette composante et si tous les arcs induits sont étiquetés par la même proposition.

Ces optimisations sont prises en compte en utilisant la représentation par des BDD (voir section 2.6.2).

2.3.3.1 Simulations

Dans un automate de Büchi généralisé étiqueté sur les états, un état s_2 simule un état s_1 si
– les étiquettes (ensemble des ensembles de propositions atomiques acceptés) de s_1 sont incluses dans celles de s_2,
– s_2 appartient au moins aux ensembles d'acceptation auxquels s_1 appartient,
– chaque successeur de s_1 est simulé par un successeur de s_2.
Cette relation de simulation permet de supprimer des transitions. Par exemple s'il existe un état s_3 dont les successeurs $\delta(s_3) = \{s_1, s_2\}$ sont tels que s_2 simule s_1, alors la transition vers s_1 peut être supprimée ($\delta(s_3) = \{s_2\}$) sans changer le langage reconnu par l'automate.

Cette relation peut aussi être définie dans l'autre sens, en travaillant sur δ^{-1} et en prenant garde aux états initiaux [60, 6].

2.3.4 Synthèse

Le tableau 2.3 (page 28) compare les six algorithmes principaux présentés : GPVW, LTL2AUT, Wring, Couvreur/FM, LTL2BA, et Couvreur/LaCIM. LTL2BÜCHI ne semble rien apporter par rapport à Couvreur/FM.

A priori, Couvreur/FM et LTL2BA semblent être deux approches prometteuses. Elles produisent toutes deux un automate étiqueté sur les transitions. L'avantage de Couvreur/FM est que la méthode est simple, qu'elle peut fonctionner à la volée, et que bon nombre de simplifications sont prises en compte automatiquement par les BDD. Néanmoins, comme il a été dit section 2.3.1.6, il semble qu'il devrait être possible de l'améliorer encore.

2.4 *Emptiness check* et recherche de contre-exemple

Les ω-automates sont des structures finies reconnaissant des mots infinis. Un tel mot ne peut être accepté que s'il existe un circuit dans l'automate, et que ce circuit satisfait les conditions d'acceptation de l'automate.

Le problème de satisfaction d'un automate de Büchi se réduit donc à la recherche d'une composante fortement connexe qui soit accessible depuis l'état initial, et qui vérifie les conditions d'acceptation de l'automate.

En *model checking*, un automate traduisant la négation d'une propriété f est synchronisé avec l'automate représentant le modèle à vérifier. La satisfaction de cet automate « produit » indique l'existence d'un contre-exemple de la propriété à vérifier, c'est-à-dire une exécution du système qui ne satisfait pas f.

Les méthodes de recherche de composantes fortement connexes peuvent être séparées en deux classes : les approches globales, qui travaillent sur l'automate dans sa totalité, et les approches *à la volée* qui sont combinées à la construction de l'automate synchronisé. Le but de ces dernières approches est de ne construire que la partie *utile* de l'automate (i.e. la construction est abandonnée dès qu'un contre exemple est obtenu).

2.4.1 Composantes fortement connexes

L'approche traditionnelle utilisée en recherche de composantes fortement connexes est l'algorithme de Tarjan [1, section 5.5]. Il s'agit d'un algorithme de recherche en profondeur, linéaire par rapport à la taille de l'automate. Malheureusement, il doit considérer chaque état individuellement, ce qui se révèle impraticable lorsque l'automate est immense.

Des algorithmes dits *symboliques* tentent de venir à bout du problème en ne considérant plus les états individuellement mais de façon ensembliste. Cela suppose que l'automate soit représenté sous une forme avec laquelle on puisse calculer l'ensemble des successeurs (et prédécesseurs) d'un ensemble d'états, comme par exemple avec un BDD (section 2.6.2.3).

Ravi et al. [56] effectue une comparaison des différents algorithmes symboliques de recherche de composantes fortement connexes. La plupart de ces algorithmes partent de l'ensemble des états et le raffinent jusqu'à obtenir une *coque* de composantes fortement connexes vérifiant les conditions d'acceptation, c'est-à-dire un ensemble d'états *contenant* au moins une telle composante (mais peut-être aussi d'autres états superflus). Si cette coque contient un état accessible de l'automate, alors la formule est satisfiable.

La complexité de tels algorithmes s'exprime en nombre d'*étapes symboliques* c'est-à-dire en nombre d'appels des fonctions successeurs et prédécesseurs. Elle va de $O(n^2)$ pour les algorithmes les plus connus à $O(n \log n)$ [7], où n est le nombre d'états de l'automate. Il s'agit d'une complexité en pire cas. Bien qu'elle soit plus grande que celle des algorithmes manipulant les états explicitement (comme Tarjan), la pratique montre que ces algorithmes convergent plus rapidement.

Les comparaisons effectuées par Ravi et al. [56] entre les différents algorithmes symboliques montrent qu'une meilleure complexité théorique ne se traduit pas toujours par un gain de temps en pratique. D'autre part, un algorithme doit aussi être jugé sur la qualité des résultats qu'il délivre. En pratique on aime obtenir un exemple lorsqu'une formule (au hasard, la négation d'une propriété à prouver) est satisfiable — tous les algorithmes comparés ne sont pas égaux à ce niveau. Les performances (donc le choix) des différents algorithmes dépendent aussi du types de composantes connexes rencontrées. Ce dernier constat a motivé la création d'un algorithme hybride [74] qui adapte ses calculs au type (fort, faible, terminal[12]) des composantes fortement connexes manipulées.

2.4.2 Tarjan *à la volée*

L'algorithme de Tarjan, déjà cité, permet d'énumérer les composantes fortement connexes maximales (au sens de l'inclusion) d'un graphe en effectuant une simple recherche en profondeur[13].

L'algorithme place les états visités sur une pile, dans l'ordre de leur visite, et associe deux valeurs à chaque état du graphe : DFNUMBER, un numéro d'ordre affecté lors du parcours en profondeur, et LOWLINK, un pointeur vers un état plus petit (au sens du numéro d'ordre) dans la même composante connexe tel qu'on ait LOWLINK=DFNUMBER à la racine des composantes fortement connexes. Le cœur de l'algorithme réside dans le calcul de LOWLINK.

Outre les problèmes liés d'une part à la manipulation explicite des états et d'autre part à la somme des informations supplémentaires qui doivent être stockées pour chaque état, l'algorithme de Tarjan va calculer des composantes fortement connexes *maximales*, ce qui demande plus d'efforts que la recherche de composantes fortement connexes simples (non-maximales) suffisantes au *model checking*.

Couvreur [20] propose une variation de cet algorithme, adaptée au *model checking* car s'arrêtant dès qu'une composante fortement connexe (pas nécessairement maximale) accessible et vérifiant des conditions d'acceptation de Büchi **généralisées** est trouvée. D'autre part, les composantes fortement connexes maximales non acceptables peuvent être *supprimées* de la mémoire. Cela rend la méthode utile dans une vérification à la volée : l'automate produit est construit au fur et à mesure des besoins de l'algorithme de recherche de composantes fortement connexes, et les morceaux de l'automate qui se révèlent être des *composantes mortes* sont détruits progressivement. La seule information importante à conserver à propos de ces états est le fait qu'ils ont été visités. Ceci peut se faire avec une table de hachage. (Par contre l'algorithme n'est pas compatible avec les mémoires d'état *à perte* présentées sections 2.6.1.1 et 2.6.1.2.)

Lorsque l'algorithme trouve une composante fortement connexe acceptable, la seule information disponible permettant éventuellement de reconstruire une exécution acceptée est la pile de composantes fortement connexes traversées (représentées par leurs racines).

2.4.3 Double recherche en profondeur

Une façon de détecter des composantes fortement connexes acceptantes (dans un automate de Büchi non-généralisé) est de faire une double recherche en profondeur. Une première recherche permet de trouver les états d'acceptation accessibles depuis l'état initial. Cette première recherche conserve la liste de tous les états visités dans une table de hachage (telles que celles décrites section 2.6.1). Cette table de hachage est vidée une fois la première recherche terminée, puis elle est réutilisée lors d'une nouvelle recherche en profondeur lancée à partir de chacun des états acceptants trouvés, pour vérifier s'il existe un cycle reliant l'un de ces états à lui-même. Ces multiples recherches peuvent partager la même table de hachage [19], ce qui fait qu'elles peuvent être vues (du point de vue de la complexité) comme une seule recherche en profondeur.

L'inconvénient de l'algorithme précédent est que la séparation des deux recherches (accessibilité, puis circuit) empêche d'exhiber un exemple de séquence acceptée. Courcoubetis et al. [19] proposent d'imbriquer ces deux recherches en profondeur, afin de pouvoir exhiber le chemin parcouru à tout moment (en particulier au moment où l'on découvre un circuit autour d'un état d'acceptation...). Cette imbrication requiert l'emploi de deux tables de hachage : une pour retenir les états visités par chacune des recherches en profondeur. Ce second algorithme utilise donc deux fois plus de place.

Godefroid et Holzmann [33] montrent comment réduire l'encombrement lié à l'imbrication des deux recherches à seulement deux bits par états. L'idée est simplement de n'utiliser qu'une unique table de hachage pour les deux recherches, mais d'associer à chaque clef (état) de cette table une valeur sur deux bits. Le premier bits indique si l'état a été visité par la première recherche (accessibilité), et le second si l'état a été visité par la seconde recherche (circuit). Ce nouvel algorithme a été baptisé *magic search*.

[12] *Grosso modo*, une composante fortement connexe est dite *faible* si tous ses cycles sont acceptants ; autrement elle est *forte*. Une composante faible est *terminale* si tous les arcs quittant cette composante amènent vers une autre composante terminale. Toutes les exécutions qui atteignent une composante fortement connexe terminale sont nécessairement acceptés.

[13] De façon générale, un seul parcours en profondeur ne suffit pas à couvrir tout le graphe. Tarjan effectue donc plusieurs parcours successifs à partir des états non atteints par les précédents. Cependant, en *model checking* nous ne nous intéressons qu'aux composantes fortement connexes accessibles à partir de l'état initial, et le premier parcours suffit.

2.4.4 Synthèse

Pour résumer, le *magic search* est un algorithme linéaire par rapport au nombre d'états (mais les états sont visités deux fois chacun), et utilisant au moins $n(2+\log n)$ bits de mémoire (n est le nombre d'états visités jusqu'à ce qu'on trouve un chemin accepté). Les conditions d'acceptation vérifiées sont les conditions classiques de Büchi. Cette technique permet aussi l'application d'optimisations probabilistes telles que celles décrites sections 2.6.1.1 et 2.6.1.2.

L'algorithme de Couvreur/FM est lui aussi linéaire par rapport au nombre d'états (mais contrairement au *magic search*, il ne visite chaque état qu'une fois). Il utilise au moins $2n \log n$ bits de mémoire pour retenir les états visités et les numéroter, plus une pile de composantes fortement connexes traversées, ainsi qu'une pile de conditions vérifiées sur les arcs entre ces composantes fortement connexes. L'encombrement de ces deux dernières piles n'est pas évident à établir. Dans tous les cas la mémoire utilisée est nettement plus importante que lors du *magic search* et ne peut être réduite avec les techniques données sections 2.6.1.1 et 2.6.1.2. En revanche, le point fort de cet algorithme est qu'il permet de vérifier directement des conditions d'acceptation de Büchi généralisées. Cela évite l'utilisation d'une étape de dégénéralisation qui d'une part doit connaître tout l'automate — impossible de travailler à la volée — et d'autre part multiplie sa taille par le nombre d'ensembles d'acceptation.

Les algorithmes symboliques permettent eux aussi de vérifier des conditions d'acceptation de Büchi généralisées, mais doivent connaître tout l'automate (dont la relation de transition est représentée par un BDD) et ne permettent donc pas un travail à la volée. Leur complexité est difficile à comparer aux algorithmes précédents puisqu'elle est exprimée en nombre d'opérations ensemblistes.

Le tableau 2.7 tente de résumer ces différents points.

algorithme	complexité en pire cas	conditions vérifiées	à la volée	optimisations probabilistes	encombrement
magic search	$O(N)$ opérations	Büchi	oui	possibles	$n(2+\log n)$
Couvreur/FM	$O(N)$ opérations	Büchi généralisé	oui	impossibles	$> 2n \log n$
algorithmes symboliques	$O(N \log N)$ à $O(N^2)$ opérations symboliques	Büchi généralisé	non	impossible	?

TAB. 2.7 – Résumé des différents algorithmes de recherche de composantes fortement connexes. N est la taille de l'automate. $n \leq N$ est le nombre d'états visités par l'algorithme avant de trouver un chemin accepté.

2.5 Hypothèses d'équité

Il est parfois utile de modéliser certaines hypothèses qui sont faites sur les systèmes que l'on cherche à vérifier. Par exemple considérons les deux processus décrits figure 2.16(a). Chaque processus possède deux états, et va passer alternativement de l'un à l'autre. Si l'on cherche à modéliser, non pas chaque processus séparément, mais le système formé de ces deux processus, on décrira plutôt le système présenté figure 2.16(b) où tous les entrelacements possibles des exécutions de ces deux processus sont pris en compte. D'un point de vue automate, cela revient à faire le produit (asynchrone) des deux automates.

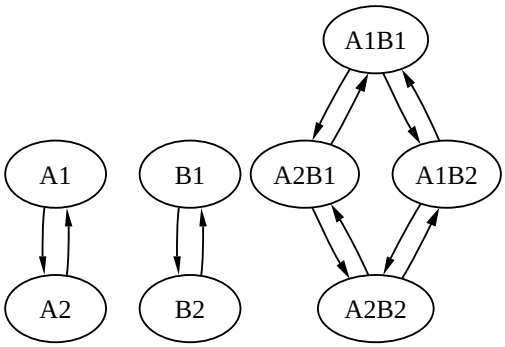


FIG. 2.16 – (a) Deux processus. (b) Le système composé des deux processus.

Ce *système produit* contient des exécutions telles que $A_1B_1, A_2B_1, A_1B_1, A_2B_1, \ldots$ dans lesquelles le processus B n'évolue pas. Sur un système multi-tâches dont le séquenceur est équitable, ou simplement si les processus tournent sur des machines différentes, il n'est pas possible qu'un processus soit bloqué de cette façon. Lors du *model checking*, on souhaite parfois ignorer ces séquences, pour ne vérifier les propriétés du système que sur les seules séquences d'exécutions qui ont un sens.

Les hypothèses d'équité peuvent être vues comme des filtres qui suppriment ce genre de séquences impossibles.

Il existe plusieurs façons de représenter ces hypothèses d'équité. Une première possibilité est de les représenter par des conditions d'acceptation [68]. Par exemple on pourrait associer aux processus A et B les conditions de Büchi généralisées $\mathcal{F}_A = \{\{A_1, A_2\}\}$, $\mathcal{F}_B = \{\{B_1, B_2\}\}$. Ces conditions n'apportent rien aux processus A et B pris séparément. Par contre, il est possible d'en déduire les conditions de Büchi généralisées associées au système produit par distribution : $\mathcal{F} = \{\{A_1B_1, A_2B_1\}, \{A_1B_2, A_2B_2\}, \{A_1B_1, A_1B_2\}, \{A_2B_1, A_2B_2\}\}$. Cet ensemble d'acceptation rejette l'exécution citée dans laquelle B n'évoluait pas (car elle ne traverse aucun état du sous-ensemble $\{A_1B_2, A_2B_2\}$).

Dans ce cas le système est vu comme un automate de Büchi généralisé, et doit ensuite être synchronisé avec l'automate de la négation de la formule à prouver. Ce produit synchronisé doit prendre garde à bien respecter les conditions d'acceptation de chaque automate.

Une autre approche consiste à représenter les hypothèses d'équité sous la forme d'une formule logique, de transformer cette formule en un automate que l'on synchronise avec le système pour en filtrer les exécutions non équitables [33]. Par exemple L'équité supposée dans notre exemple correspond à la formule $eq = \mathsf{GF}\, A_1 \wedge \mathsf{GF}\, A_2 \wedge \mathsf{GF}\, B_1 \wedge \mathsf{GF}\, B_2$. Vérifier une formule φ sur notre système S sous l'hypothèse eq revient donc à effectuer l'*emptiness check* de $A_S \otimes A_{eq} \otimes A_{\neg\varphi}$.

Les formules du type GF jouent un rôle important dans la spécification d'hypothèses d'équité. C'est pourquoi Couvreur [20] traite GF comme un opérateur particulier pour en améliorer la traduction. (Un automate tel que notre A_{eq} serait représenté par un seul nœud.)

2.6 Techniques d'implémentation

2.6.1 Mémoire d'états

Lorsque l'on construit un produit synchronisé afin de tester s'il est vide, il est inutile de stocker l'automate produit dans sa totalité. Il suffit de ne conserver que les informations strictement nécessaires au test.

Considérons $A = A_M \otimes A_\varphi$, l'automate produit du système et de la formule à vérifier. Cet automate possède $|S| = |S_M| \times |S_\varphi|$ états, mais en pratique tous ces états ne sont pas accessibles. Nous noterons $n \leq |S|$ le nombre d'états accessibles de A. n est inconnu a priori, il dépend du système et de l'automate généré pour la formule à vérifier.

Chaque état peut être représenté par une valeur sur $\log_2 |S|$ bits (c'est-à-dire numéroté par des valeurs de $1 \ldots |S|$). Il existe deux façons simples de retenir qu'un état a été visité. Soit on utilise ces valeurs comme indices dans un tableau de bits (le bit est mis à 1 lorsque l'état été visité), soit on stocke ces valeurs dans un tas (la valeur est ajoutée au tas lorsque l'état est visité).

Les temps d'accès de ces deux représentations mémoire sont différents (constant pour la première, logarithmique pour la seconde), cependant, c'est l'occupation mémoire qui fait office de goulet en *model checking*. L'approche *tableau de bits* utilise $|S|$ bits de mémoire tandis que le *tas* occupera $n \times \log_2 |S|$ bits. Le choix d'une représentation ou de l'autre devrait donc se faire en fonction de n, qui est malheureusement inconnu.

En pratique, les deux approches utilisent trop de mémoire, et ont donné lieu à plusieurs techniques plus évoluées, dans lesquelles on accepte d'*oublier* certains états en échange d'une occupation mémoire plus concise.

2.6.1.1 Hachage d'états sans détection de collision

Le hachage d'état est une approche *tableau de bits*, dans laquelle on compacte la représentation d'un état pour réduire la taille du tableau. C'est-à-dire qu'on va utiliser une fonction de hachage f pour faire passer les identifiants des états de $\{1, \ldots, |S|\}$ à $\{1, \ldots, m\}$, avec $m \ll |S|$. De cette façon le tableau des états visités n'occupera plus que m bits de mémoire.

Naturellement des collisions peuvent survenir : deux états s_1 et s_2 peuvent être hachés vers la même valeur ($f(s_1) = f(s_2)$). La conséquence d'une telle collision est qu'un état non visité pourra être considéré comme un état déjà visité, dissimulant ainsi une partie du système au *model checker*. Des erreurs pourront donc passer inaperçues.

Le résultat du *model checker* doit alors être interprété de façon probabiliste : soit une erreur a été détectée et l'on est sûr que le système 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. Ce type de vérification est plus utile au débuggage qu'à la vérification pure [19].

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 [78], Holzmann [40] comparent ces différentes méthodes d'un point de vue probabiliste.

Dans tous les cas, même si m est plus petit que $|S|$, il doit être choisi suffisamment grand pour limiter les nombre de collisions. Wolper et Leroy [78] montrent 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 approche est appelée *hashcompact*.

2.6.1.2 Cache d'états

Le cache d'états est une approche dans laquelle les états visités sont stockés dans un tas, mais un tas dont la taille est limitée. Ce tas est alors appelé un *cache*. Lorsque le cache est plein et qu'on souhaite y ajouter un nouvel état, on en retire un autre. Cela signifie que les états retirés du cache ne sont plus considérés visités, et pourront être visités à nouveau.

Bien que cette méthode permette de borner la taille de la mémoire utilisée, elle augmente les temps de calculs de façon significative. Dès qu'un état est accessible par plusieurs chemins, on court le risque de le réexplorer plusieurs fois, lui et tous ses successeurs, s'il est sorti du cache entre deux visites.

En introduisant des connaissances supplémentaires sur le système, comme l'indépendance de certaines transitions, Godefroid et al. [32] montrent comment limiter le nombre de chemins conduisant à des états déjà visités, réduisant ainsi les accès à des états sortis du cache.

Il est bon d'insister sur le fait qu'un cache d'états ne laissera pas des erreurs passer inaperçues : à la différence du hachage d'états, deux états ne sont jamais confondus. La technique est certes beaucoup plus coûteuse en temps, mais elle est sûre.

2.6.2 BDD

Beaucoup de problèmes peuvent être réduits à une séquence d'opérations sur des fonctions booléennes. Les représentations utilisées classiquement sont la forme normale disjonctive (FND), la forme normale conjonctive (FNC), ou même la table de vérité.

Les BDD (*Binary Decision Diagrams*) sont des structures de données particulières permettant une représentation et une manipulation très efficaces de fonctions booléennes. La forme la plus commune de BDD a été introduite par Bryant [8] et a ensuite trouvé de nombreuses applications au *model checking* [10, 54, 55, 14] car elle permet de représenter la relation de transition d'un système de transition sous une formule ensembliste.

Outre leur simplicité, et leur adéquation à de nombreux problèmes, le principal avantage des BDD est le faible coût des opérations logiques. Le tableau 2.8 compare les complexités de différentes formes normales.

Opération	Résultat	FND	FNC	BDD
Réduction	$\mathcal{G}_f$ forme canonique	n/a	n/a	$\mathcal{O}(\|\mathcal{G}_f\| \times \|log(\mathcal{G}_f\|)$
Opération binaire	$f_1 \langle op \rangle f_2$	exp	exp	$\mathcal{O}(\|\mathcal{G}_{f_1}\| \times \|\mathcal{G}_{f_2}\|)$
Composition	$f_{1_{x_i=f_2}}$	exp	exp	$\mathcal{O}(\|\mathcal{G}_{f_1}\|^2 \times \|\mathcal{G}_{f_2}\|)$
Négation	$\neg f$	exp	exp	$\mathcal{O}(1)$

TAB. 2.8 – Complexité en temps suivant la représentation.

2.6.2.1 Représentation d'une formule booléenne

Pour donner une forme normale à une expression booléenne, Andersen [2] commence par définir un opérateur If-then-else comme suit : $x \longrightarrow y_0, y_1 =_{def} (x \wedge y_0) \vee (\neg x \wedge y_1)$. Une forme normale *If-Then-Else* (notée INF) est une expression booléenne construite en considérant les constantes 0 et 1 et en appliquant l'opérateur if-then-else sur des variables uniquement. Pour une expression t donnée on peut alors générer une INF. En utilisant l'expansion de Shannon ($t = x \longrightarrow t[1/x], t[0/x]$) sur un ordre choisi des variables de l'expression, on retrouve un ensemble de sous expressions permettant de voir l'expression initiale comme un *arbre de décision*.

Exemple 1. *Pour* $t = (x_1 \Leftrightarrow x_2) \wedge (x_3 \Rightarrow x_4)$ *avec l'ordre sur les variables de* t *défini comme suit :* $x_1 < x_2 < x_3 < x_4$*, on obtient les sous expressions suivantes :*
– $t = x_1 \rightarrow t_1, t_0$
– $t_0 = x_2 \rightarrow 0, t_{00}$ *et* $t_1 = x_2 \rightarrow t_{11}, 0$
– $t_{00} = x_3 \rightarrow t_{001}, t_{000}$ *et* $t_{11} = x_3 \rightarrow t_{111}, t_{110}$
– $t_{000} = x_4 \rightarrow 1, 1$ *et* $t_{001} = x_4 \rightarrow 1, 0$,
– $t_{110} = x_4 \rightarrow 1, 1$ *et* $t_{111} = x_4 \rightarrow 1, 0$.
Chaque sous-expression représente un sommet de l'arbre de décision. La figure 2.17(a) montre cet arbre.

Un BDD est un graphe orienté sans circuit possédant deux types de sommets.
– si un sommet v est terminal alors il est étiqueté par $value(v) \in \{0, 1\}$,
– sinon v est non-terminal et il est étiqueté par un variable $val(v)$, il a deux arcs donnés par $low(v)$ et $high(v)$. ($low(v)$ correspond à la partie *else* et $high(v)$ à la partie *then*.)
Un BDD est dit *ordonné* (noté OBDD) si pour tout chemin du graphe les variables apparaissant dans le même ordre. On dit qu'un (O)BDD est *réduit* (noté ROBDD) lorsqu'on a supprimé les sommets inutiles (non redondance du test : $low(\alpha) \neq high(\alpha)$) et fusionné les sous-graphes isomorphes (unicité).

Par exemple le ROBDD de la figure 2.17(b), obtenu à partir du BDD de la figure 2.17(a), correspond aux sous-expressions suivantes :
– $t = x_1 \rightarrow t_1, t_0$
– $t_0 = x_2 \rightarrow 0, t_{00}$ et $t_1 = x_2 \rightarrow t_{00}, 0$,
– $t_{00} = x_3 \rightarrow t_{001}, 1$,
– $t_{001} = x_4 \rightarrow 1, 0$.
En pratique lorsque l'on parle de BDD on fait référence à des ROBDD ou à d'autres formes plus évoluées.

Chaque sommet d'un BDD définit inductivement une expression booléenne. En pratique pour maximiser le partage de l'information, nous utiliserons un BDD pour représenter un *forêt* de fonctions, dont on espère qu'elles partageront des sous-graphes. Dans ce cas les complexités listées dans le tableau 2.8 ne sont plus tout à fait exactes. Par exemple la négation d'une formule représentée isolément dans le BDD peut se faire en temps constant simplement en permutant les feuilles 0 et 1 ; lorsque plusieurs formules partagent ces feuilles ce n'est plus possible.

Bryant [8] montre l'existence et l'unicité du ROBDD associé à une fonction booléenne donnée pour un ordre total sur les variables prédéfini. La recherche d'un ordre minimisant la taille du BDD est un problème NP-complet. Certaines heuristiques donnent d'assez bons résultats, par exemple celle qui consiste à placer les variables dans l'ordre où elles sont rencontrées lors d'un parcours profondeur de la formule.

Les BDD peuvent aussi être vus comme une forme d'automate fini déterministe [15]. Une fonction booléenne d'arité n peut être vue comme un ensemble de mots sur $\{0, 1\}^n$ qui représente les affectations de variables satisfaisant la fonction booléenne. Puisque le langage est fini, il est régulier. Donc il existe un automate fini et minimal qui accepte cet ensemble. Cet automate offre une représentation canonique de la fonction booléenne. Les constructions standard avancées par la théorie élémentaire des automates peuvent être utilisées pour implémenter les opérations sur le langage. Par exemple, $\wedge$ correspond à l'intersection d'ensemble.

2.6.2.2 Représentation d'un ensemble

Tout ensemble $\mathcal{H} \subseteq \{0, 1\}^n$ peut être représenté par sa fonction caractéristique

$$1_{\mathcal{H}}(b_1, b_2, \ldots, b_n) = \begin{cases} 1 & \text{si } (b_1, b_2, \ldots, b_n) \in \mathcal{H}, \\ 0 & sinon. \end{cases}$$

Une façon simple de construire $1_{\mathcal{H}}$, est de prendre la disjonction de tous les éléments de $\mathcal{H}$ (vus comme des conjonctions).

Un sous-ensemble de $\{0, 1\}^n$ peut donc être représenté en BDD par sa fonction caractéristique.

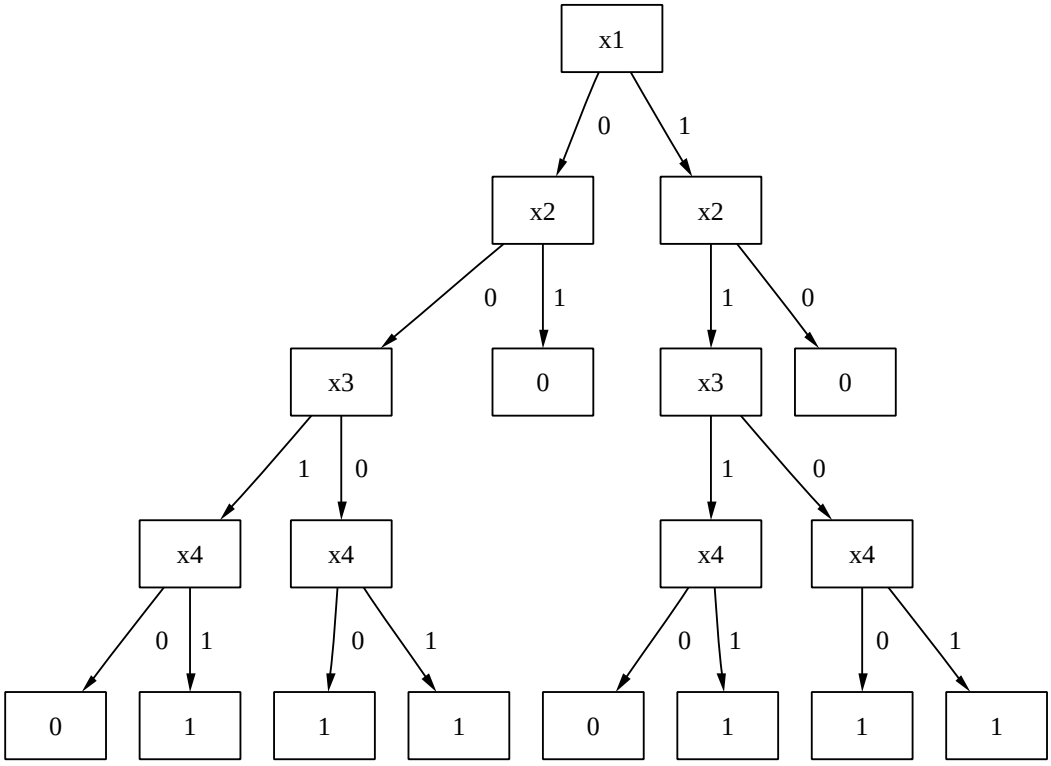
2.6.2.3 Représentation d'une structure de Kripke

Toute relation sur un domaine fini peut être représentée par un BDD de la manière suivante [15].

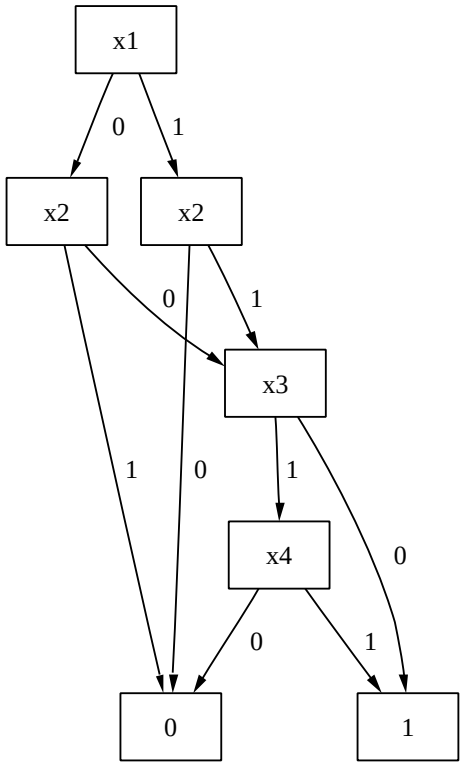
Si Γ est une relation de dimension n sur $\{0, 1\}$ alors Γ est représenté en BDD par sa fonction caractéristique

$$(f_\Gamma(x_1, x_2..., x_n) = 1) \longleftrightarrow \Gamma(x_1, x_2, ..., x_n).$$

Dans un cas plus général, on considère un domaine $\mathcal{D}$ de dimension 2^m. Pour représenter Γ par un BDD, on affecte les valeurs de $\mathcal{D}$ par une bijection $\Phi : \{0, 1\}^m \longrightarrow \mathcal{D}$. Nous retrouvons alors une fonction booléenne $\vec{\Gamma}$ d'arité $m \times n$



(a)



(b)

FIG. 2.17 – (a) OBDD et (b) ROBDD pour $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Rightarrow x_4)$.

telle que

$$\vec{\Gamma}(\vec{x_1}, \vec{x_2}, ..., \vec{x_n}) = \Gamma(\Phi(\vec{x_1}), \Phi(\vec{x_2}), ..., \Phi(\vec{x_n})).$$

avec $\vec{x_i}$ un vecteur de $\{0,1\}^m$ pour $i \in [0,n]$. Ainsi, on peut déterminer la représentation de Γ en BDD par la fonction caractéristique de $\vec{\Gamma}$.

La fonction de transition d'une structure de Kripke peut être interprétée comme une relation de $\mathcal{R} \subseteq S \times S$.

Pour encoder les états, on considère comme précédemment une bijection $\Phi : \{0,1\}^m \longrightarrow S$ (en supposant que l'on a 2^m états). Puisque toute affectation de valeurs est donnée par Φ, la fonction caractéristique de S est le BDD pour 1.

Pour la relation de transition $\mathcal{R}$ on garde la même fonction d'encodage, mais on travaille avec deux ensembles de variables booléennes : un pour représenter l'état entrant et l'autre pour représenter l'état sortant d'une transition de $\mathcal{R}$. Soit $\vec{\mathcal{R}}(\vec{x}, \vec{x'})$ la relation de transition encodant $\mathcal{R}$. La fonction caractéristique de $\vec{\mathcal{R}}$ représente $\mathcal{R}$.

Sur un exemple, la relation de transition du système modélisant le processus présenté figure 2.16(b) (page 34) possède une fonction caractéristique à huit variables. $T(v_{11}, v_{12}, v_{21}, v_{22}, v'_{11}, v'_{12}, v'_{21}, v'_{22})$. Les variables v_{ij} indiquent si l'état A_iB_j est occupé à l'instant présent. Les variables v'_{ij} indiquent si ces mêmes états peuvent être occupés à l'instant suivant. En notant $\vec{v} = (v_{11}, v_{12}, v_{21}, v_{22})$, $T(\vec{v}, \vec{v'})$ est vrai si les transitions font passer de l'ensemble d'états indiqués par $\vec{v}$ à celui indiqué par $\vec{v'}$.

L'état du système A_1B_1 peut être représenté par $s = v_{11} \wedge \neg v_{12} \wedge \neg v_{21} \wedge \neg v_{22}$. Trouver les successeurs de s par T revient à calculer $T \wedge s$, puis à supprimer toutes les variables de type v_{ij}, et enfin à renommer les v'_{ij} en v_{ij}. Cette opération notée $(\exists \vec{v}.T \wedge s)[\vec{v}/\vec{v'}]$ retourne une nouvelle formule sur $\vec{v}$. Ici, $s_2 = \neg v_{11} \wedge v_{12} \wedge v_{21} \wedge \neg v_{22}$. C'est-à-dire que les successeurs sont A_1B_2 et A_2B_1.

Il faut noter que s comme s_2 sont des ensembles d'états. Cette façon de calculer le successeur fonctionne de façon ensembliste. Par exemple le calcul du successeur de s_2 retournera $s_3 = v_{11} \wedge v_{12} \wedge v_{21} \wedge v_{22}$, c'est-à-dire tous les états.

Ce travail ensembliste fait la force des BDD en *model checking*, car il évite d'énumérer tous les états explicitement. L'introduction des BDD dans le domaine du *model checking* a permis de repousser la limite du nombre d'états traitables. Burch et al. [10] fut le premier à rapporter un bond de 10^8 vers 10^{20} états traitables en utilisant les BDDs. L'utilisation de variantes des BDD permet de gérer encore plus d'états. Les articles récents citent des valeurs de l'ordre de 10^{100} [14].

2.6.3 Optimisations du modèle

Il existe plusieurs façons de réduire la taille du modèle (c'est-à-dire celle du graphe des états accessibles).

Dans la méthode des *stubborn sets* [66] et celle des *sleep sets* [32] on considère que la vérification de certaines propriétés du modèle ne requièrent pas la génération de toutes les permutations possibles de franchissements de transitions indépendantes.

D'autres techniques, telles que l'analyse des symétries du modèle, permettent de créer des classes d'équivalence de marquages [36, 35].

Prod [73] est un générateur de graphe de marquage implémentant plusieurs de ces techniques et un *model checker* correspondant.

2.7 Vérification des traducteurs de formules LTL

La vérification d'un système avec une formule n'est pas une opération triviale : aucun des différents algorithmes présentés précédemment ne coule de source.

L'une des phases les plus complexes, à la fois conceptuellement et algorithmiquement, est la transformation d'une formule LTL en un ω-automate. La quête de l'automate le plus petit (motivée par l'impact de la taille tableau sur celle du produit synchronisé) conduit les auteurs à inventer de nouvelles optimisations qui, tout en réduisant le nombre d'états, complexifient les algorithmes.

De tels algorithmes restent démontrables mathématiquement, mais deviennent difficiles à implémenter. D'autre part, une implémentation cherchera à optimiser à la fois l'espace et le temps utilisé, c'est-à-dire s'éloignera de l'algorithme tel qu'il a été formalisé.

Comment avoir confiance en de telles implémentations ? Comment les tester ? La littérature sur le sujet est quasi nulle.

Certains auteurs, comme Daniele et al. [23], testent leurs algorithmes en utilisant une batterie de formules générées aléatoirement. Cet ensemble de formules n'est utilisé qu'à des fins statistiques, pour comparer la taille des automates

générés selon plusieurs critères. Ceci ne donne aucune idée sur la validité de l'implémentation, néanmoins l'idée de générer des formules est une première piste.

Tauriainen [62], Tauriainen et Heljanko [64] proposent quatre tests pour exercer de façon automatique un traducteur de formule LTL en ω-automate. Ces tests sont basés sur des propriétés simples des langages acceptés par les automates. Naturellement, ces tests ne servent qu'à trouver des erreurs dans les implémentations, ils ne montrent *pas* qu'une implémentation est correcte.

1. Soit φ une formule à tester. Alors $\mathscr{L}(A_\varphi \otimes A_{\neg\varphi}) = \emptyset$. Autrement dit, le produit synchronisé entre l'automate correspondant à une formule et l'automate correspondant à la négation ne doit accepter aucun mot.
2. De même, on devrait avoir $\mathscr{L}(A_\varphi \cup A_{\neg\varphi}) = AP^\omega$, c'est-à-dire que tout mot infini sur l'alphabet AP est reconnu par A_φ ou $A_{\neg\varphi}$. En pratique, cette vérification peut se faire en complémentant l'automate union, et en vérifiant $\mathscr{L}(\neg(A_\varphi \cup A_{\neg\varphi})) = \emptyset$. Malheureusement, la négation d'un automate de Büchi est une opération de coût spatial exponentiel [70].
3. Si l'on possède plusieurs implémentations transformant une formule LTL, on peut les comparer comme suit. Pour chaque traducteur i, on construit l'automate A^i_φ correspondant à la formule φ. Chacun de ces automates est synchronisé avec un même système, $A_S \otimes A^i_\varphi$, puis on vérifie si le langage reconnu est nul. Les résultats doivent être identiques pour l'ensemble des automates.

   Ce test peut être amélioré en effectuant un produit synchronisé global (qui ne se limite pas aux seuls états accessibles depuis l'état initial) et en considérant tour à tour chaque état du système comme un état initial. Cela revient à effectuer ce test avec de nombreux systèmes différents, mais en n'ayant effectué qu'un seul produit synchronisé pour chaque type de traducteur.
4. De façon similaire, pour un système A_S donné, et deux automates A_φ et $A_{\neg\varphi}$ obtenus par le même algorithme ou deux algorithmes différents, on doit vérifier que si $A_S \otimes A_\varphi$ est nul, alors $A_S \otimes A_{\neg\varphi}$ ne l'est pas, et vice-versa.

   Comme précédemment, il est possible d'effectuer des produits synchronisés globaux, puis d'effectuer cette vérification en considérant tour à tour chaque état de A_S comme état initial.

Lorsque l'un de ces tests échoue, c'est que l'un des deux automates impliqués est faux. Malheureusement il est impossible *a priori* de désigner l'automate fautif. Par contre on peut extraire de chaque erreur une séquence permettant d'analyser l'erreur. Par exemple si le premier test échoue, il exhibe une séquence qui est acceptée à la fois par A_φ et $A_{\neg\varphi}$. Cette séquence peut être comparée manuellement à φ pour trouver l'automate fautif.

Tauriainen [62] note que ces séquences témoins sont des structures de Kripke séquentielles (Cf. section 2.1.3) qui peuvent être vérifiées très simplement avec des techniques comparables à celles utilisées en vérification de formules CTL [45, remarque 5.2].

Ces différents travaux ont étés concrétisés dans LBTT, un outil de vérification automatique de traducteur de formules LTL en automate de Büchi. LBTT peut s'interfacer avec de nombreux traducteurs existant ; ses premières applications ont permis de corriger des erreurs dans le traducteur de SPIN [63].

2.8 Un *model checker* connu : SPIN

Spin [39] est un *model checker* adapté aux problèmes de la vérification d'algorithmes répartis. Il utilise son propre langage pour spécifier le système à vérifier : Promela (*Process Meta Language*), un langage impératif avec des primitives de communication.

À partir de la définition en Promela du système, Spin construit des automates M_i représentant les comportements asynchrones de chaque processus. Le comportement global du système est l'automate produit $M = \prod_i M_i$. Cet automate M peut ensuite être vérifié avec des formules LTL.

La traduction d'un formule LTL en un automate de Büchi est fait avec une variation de l'algorithme GPVW (section 2.3.1.3) produisant un automate de Streett. Cet automate est ensuite converti en un automate de Büchi étiqueté sur les transitions[14]. Plusieurs optimisations d'Etessami et Holzmann [25] sont appliquées.

L'*emptiness check* du produit synchronisé est effectué avec une double recherche en profondeur (section 2.4.3).

Spin cherche à réduire l'espace d'états en utilisant des techniques de *compression d'états*, de *hachages* (section 2.6.1.1), ou d'*ordre partiel* (*Sleep set* et *Stubborn set*). Les réductions par ordre partiel [41] génèrent un espace d'états réduit considérant des *classes* de séquences d'exécutions ne modifiant pas la validité de la propriété (section 2.6.3).

Spin offre plusieurs modes de simulation d'exécution qui permettent entre autre la simulation de contre-exemples de la propriété vérifiée lorsqu'ils existent.

[14] Nous n'avons pas trouvé de description de cette traduction.

Spin a de nombreuses applications industrielles [39], et sert aussi d'implémentation de référence pour de nombreux algorithmes ou optimisations publiés chaque année. Il possède son propre *workshop* annuel.

Gastin et Oddoux [28] montrent les limites de Spin avec les deux formules suivantes.

1. Soit la formule de réponse $\mathsf{G}(q \Rightarrow \mathsf{F}\, r)$ avec η conditions d'équités :

$$\phi_\eta = \neg((\mathsf{G}\,\mathsf{F}\, p_1 \wedge ... \wedge \mathsf{G}\,\mathsf{F}\, p_\eta) \Rightarrow \mathsf{G}(q \Rightarrow \mathsf{F}\, r)) \tag{2.3}$$

   Les cas pratiques utilisent généralement $\eta > 5$. Or pour $\eta > 5$ Spin échoue, il ne produit pas l'automate de Büchi en temps et espace raisonnable.

2. Pour la formule LTL suivante

$$\psi_\eta = \neg(p_1\, \mathsf{U}(p_2\, \mathsf{U}(...\, \mathsf{U}\, p_\eta)...) \tag{2.4}$$

   Spin ne donne pas une construction de l'automate de Büchi en temps et espace raisonnable pour $\eta > 7$.

Chapitre 3

Approche choisie : automates de Büchi généralisés étiquetés sur les transitions.

Parmi les différents algorithmes présentés dans chapitre précédent, trois (Couvreur/LaCIM, LTL2BÜCHI, et Couvreur/FM) produisent des automates de Büchi généralisés étiquetés sur les transitions. Ces automates ont été baptisés TGBA (*Transition-based Generalized Büchi Automaton*) par Giannakopoulou et Lerda [30, 31] et nous réutiliserons ce nom.

Contrairement à certains automates utilisés par le passé (par exemple dans SPIN) et qui faisaient porter les propriétés à vérifier sur les arcs mais définissant les ensembles d'acceptation sur les états, les TGBA définissent leurs ensembles d'acceptation comme des ensembles des transitions. Il s'agit d'une évolution récente dans le domaine du *model checking*. Pour les raisons expliquées section 2.3.1.5, ce simple passage des étiquettes et conditions d'acceptation des états vers les transitions permet de créer des automates plus compacts lors de la traduction de formules.

D'autre part les algorithmes qui manipulent des TGBA sont compatibles avec les automates étiquetés sur les états : ces derniers peuvent en effet être interprétés comme des TGBA, en poussant les étiquettes sur les transitions sortantes ou entrantes de chaque état, sans en changer la taille. La réciproque est moins vraie : la transformation d'un TGBA quelconque en automate de Büchi généralisée étiqueté sur les états produit un automate dont chaque état correspond à une transitions du TGBA.

En choisissant d'utiliser des TGBA dans notre bibliothèque, nous avons donc choisi une représentation d'automates moderne et concise, qui nous assure toujours une compatibilité avec les automates *produits* par les algorithmes existants (notamment les algorithmes de traduction). Par contre cette représentation est incompatible avec la quasi totalité des algorithmes manipulant des automates dont les conditions d'acceptation portent sur les états (par exemple *magic search*).

Avant de détailler la bibliothèque (chapitre suivant), nous définissons les TGBA et les opérations sur ceux-ci de façon formelle.

3.1 Définition d'un TGBA

Définition 7. *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 sont des ensembles de transitions. C'est à dire un quintuplet* $A = \langle AP, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ *où*

- AP *est un ensemble fini de propositions atomiques,*
- $\mathcal{Q}$ *est ensemble finit d'états,*
- $\delta \subseteq \mathcal{Q} \times 2^{2^{AP}} \times \mathcal{Q}$ *est la relation de transition de l'automate (chaque transition étant étiquetée par une formule propositionnelle) que nous utiliserons aussi comme une fonction* $\mathcal{Q} \mapsto 2^{2^{2^{AP}} \times \mathcal{Q}}$.
- $q_0 \in \mathcal{Q}$ *est l'état initial,*
- $\mathcal{F} = \{F_1, F_2, \ldots, F_3\}$ *est un ensemble d'ensembles de transitions d'acceptation :* $\forall i, F_i \subseteq 2^{\delta}$.

3.2 Langage d'un TGBA

Définition 8. *Un ω-mot $\sigma = p_0 \cdot p_1 \cdot p_2 \cdots$ sur l'alphabet 2^{AP} est accepté par un TGBA $A = \langle AP, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ si et seulement si il existe un chemin infini*

$$q_0 \xrightarrow{P_0} q_1 \xrightarrow{P_1} q_2 \xrightarrow{P_2} \cdots$$

tel que
- *le mot est reconnu par le chemin ($\forall i, p_i \in P_i$),*
- *le chemin appartient à l'automate ($\forall i, (q_i, P_i, q_{i+1}) \in \delta$),*
- *il passe infiniment souvent par chaque ensemble d'acceptation ($\forall i, \forall F \in \mathcal{F}, \exists j \geq i : (q_j, P_j, q_{j+1}) \in F$).*

Définition 9. *Le langage de A, noté $\mathscr{L}(A)$, est l'ensemble des ω-mots acceptés par un TGBA A.*

Définition 10. *L'emptiness check d'un TGBA A est l'opération qui cherche à décider si $\mathscr{L}(A) = \emptyset$, ou de façon contraposée qui cherche à savoir s'il existe un ω-mot accepté par A.*

3.3 Traduction d'une structure de Kripke

Une structure de Kripke $KS = \langle AP, S, \delta, l, s_0 \rangle$ (cf. section 2.1.2) peut être vue comme un TGBA $\langle AP, S, \delta', s_0, \emptyset \rangle$ en poussant les propositions des états sur leurs arcs sortants, c'est-à-dire $\delta' = \{(q, \{l(q)\}, \delta(q)) \,|\, q \in S\}$, et en ne définissant aucun ensemble d'acceptation afin que tous les chemins soient acceptants.

De façon similaire, une structure de Kripke étiquetée $KS = \langle AP, E, S, \delta, l, s_0 \rangle$ (cf. section 2.1.4) peut être vue comme un TGBA $\langle AP \cup E, S, \delta', s_0, \emptyset \rangle$, en posant $\delta' = \{(q, \{l(q) \cup \{e\}\}, s) \,|\, q \in S,\ (e, s) \in \delta(q)\}$.

Des constructions similaires peuvent être utilisées pour traduire en TGBA d'autres types d'automates de Büchi. À la différence des structures de Kripke, il faudra ajuster l'ensemble des conditions d'acceptation.

3.4 Produit synchronisé

Une opération importante en *model checking* est le produit synchronisé de deux automates.

Définition 11. *Le produit synchronisé de deux automates $A_1 = \langle AP, \mathcal{Q}_1, \delta_1, q_1, \{f_1, f_2, \ldots f_n\} \rangle$ et $A_2 = \langle AP, \mathcal{Q}_2, \delta_2, q_2, \{g_1, g_2, \ldots g_m\} \rangle$ est l'automate noté $A_1 \otimes A_2 = \langle AP, \mathcal{Q}', \delta', q', \mathcal{F}' \rangle$ où*
- $\mathcal{Q}' = \mathcal{Q}_1 \times \mathcal{Q}_2$,
- $q' = (q_1, q_2)$,
- $\delta' = \{((s_1, s_2), P_1 \cap P_2, (t_1, t_2)) \,|\, (s_1, P_1, t_1) \in \delta_1, (s_2, P_1, t_2) \in \delta_2,\ \text{et } P_1 \cap P_2 \neq \emptyset\}$,
- $\mathcal{F}' = \{f'_1, f'_2, \ldots, f'_n, g'_1, g'_2, \ldots, g'_m\}$
  avec $f'_i = \{((s_1, s_2), P_1 \cap P_2, (t_1, t_2)) \in \delta' \,|\, (s_1, P_1, t_1) \in f_i\}$,
  et $g'_i = \{((s_1, s_2), P_1 \cap P_2, (t_1, t_2)) \in \delta' \,|\, (s_2, P_2, t_2) \in g_i\}$

Théorème 1. *La classe des langages reconnaissables par des TGBA est close par intersection, et*

$$\mathscr{L}(A_1) \cap \mathscr{L}(A_2) = \mathscr{L}(A_1 \otimes A_2)$$

Démonstration. Un mot $\sigma = p_0 \cdot p_1 \cdot p_2 \cdots$ de 2^{AP} est reconnut par A_1 et A_2 si et seulement si il existe un chemin acceptant ce mot dans chacun des automates.

C'est-à-dire des chemins

$$q_1^1 \xrightarrow{P_1^1} q_1^2 \xrightarrow{P_1^2} q_1^3 \xrightarrow{P_1^3} \cdots$$

$$q_2^1 \xrightarrow{P_2^1} q_2^2 \xrightarrow{P_2^2} q_2^3 \xrightarrow{P_2^3} \cdots$$

tels que

$$\begin{cases} \forall i, p_i \in P_1^i, \\ \forall i, (q_1^i, P_1^i, q_1^{i+1}) \in \delta_1, \\ \forall i, \forall F \in \{f_1, f_2, \ldots f_n\}, \exists j \geq i : (q_1^j, P_1^j, q_1^{j+1}) \in F, \end{cases}$$

et

$$\begin{cases} \forall i, p_i \in P_2^i, \\ \forall i, (q_2^i, P_2^i, q_2^{i+1}) \in \delta_2, \\ \forall i, \forall F \in \{g_1, g_2, \ldots g_m\}, \exists j \geq i : (q_2^j, P_2^j, q_2^{j+1}) \in F. \end{cases}$$

Par construction, il est donc équivalant de dire

$$\begin{cases} \forall i, p_i \in P_1^i \cap P_2^i, \\ \forall i, ((q_1^i, q_2^i), (P_1^i \cap P_2^i), (q_1^{i+1}, q_2^{i+1})) \in \delta', \\ \forall i, \forall F \in \{f'_1, f'_2, \dots f'_n, g'_1, g'_2, \dots g'_m\}, \exists j \geq i : ((q_1^j, q_2^j), (P_1^j \cap P_2^j), (q_1^{j+1}, q_2^{j+1})) \in F. \end{cases}$$

c'est-à-dire que

$$(q_1^1, q_2^1) \overset{P_1^1 \cap P_2^1}{\longrightarrow} (q_1^2, q_2^2) \overset{P_1^2 \cap P_2^2}{\longrightarrow} (q_1^3, q_2^3) \overset{P_1^3 \cap P_2^3}{\longrightarrow} \cdots$$

est un chemin acceptant de l'automate $A_1 \otimes A_2$, et il reconnaît σ. □

Chapitre 4

Spot

Spot est la bibliothèque de *model checking* que nous avons développée au cours de ce stage. Son nom est un acronyme qui signifie *Spot Produces Our Traces*. Ce chapitre détaille chacun des modules de Spot.

4.1 Structuration

La figure 4.1 présente l'architecture de Spot de façon grossière.

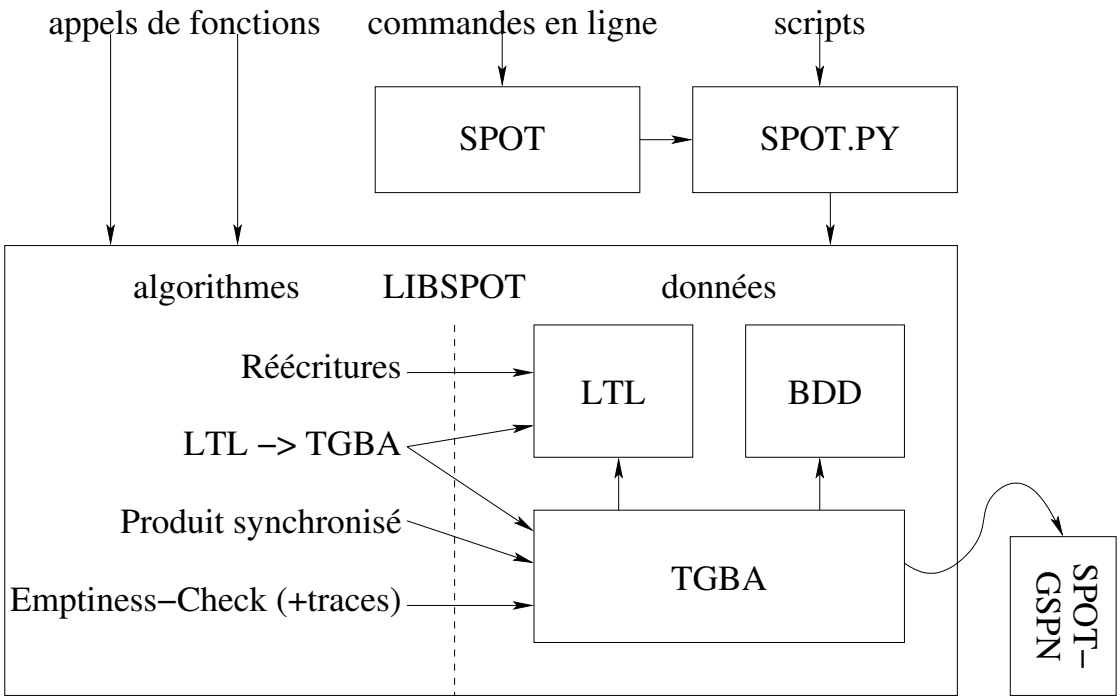


FIG. 4.1 – Structuration de Spot

La bibliothèque, `libspot`, constitue le composant principal de Spot. Elle déclare les types de donnés manipulés ainsi qu'un certains nombre d'algorithmes sur ceux-ci. Ces fonctionnalités peuvent être regroupées en trois modules :
– LTL : la gestion des formules de logique temporelle
– BDD : la gestion des diagrammes de décision binaire
– TGBA : la gestion des automates de Büchi généralisés étiquetés sur les transitions
L'interface entre Spot et GreatSPN s'intègre au niveau du modules des TGBA. Il s'agit là d'un exemple d'extension de `libspot` : l'interface est réalisée par une bibliothèque séparée qui introduit un nouveau type d'automate, manipulable par les algorithmes de `libspot`.

D'autres outils que GreatSPN doivent pouvoir être intégrés de la même façon.

`libspot`, en tant que bibliothèque C++ est utilisable directement depuis un programme C++. Elle est également accessible depuis le langage de script Python grâce à une interface dédiée nommée `spot.py` sur la figure.

À terme, l'objectif est de réaliser un outil en ligne de commande utilisant l'interface Python et celle de GreatSPN, pour vérifier une formule LTL sur un réseau de Petri. C'est la petite boîte SPOT de la figure, qui n'existe pas pour le moment.

4.2 libspot

Cette section décrit les différents modules de `libspot` introduits précédemment.

Cette bibliothèque se veut extensible : autant que faire se peut, l'utilisateur doit avoir la possibilité d'écrire ses propres algorithmes (utilisant les structures de données de Spot) ou, à l'inverse, utiliser les algorithmes de Spot sur ses propres structures de données dans le but d'optimiser le *model checking*. Ceci explique certains choix architecturaux.

4.2.1 BDD

`libspot` ne contient pas sa propre implémentation de diagrammes de décision binaire (BDD), elle utilise la bibliothèque existante BuDDy de Lind-Nielsen [48]. Le choix s'est porté sur cette bibliothèque parce qu'elle est assez récente, et qu'elle fournit une interface C++ (cette dernière permet d'écrire des formules logiques de façon plus lisible, et libère le programmeur des opérations administratives liées au comptage de références).

La version 2.2 de BuDDy, datée du 9 novembre 2002, offre un riche ensemble de fonctionnalités mais a néanmoins dû être étendue pour les besoins de Spot.

Le fonctions suivantes ont été ajoutées :

`bdd_copypairs()`

`bdd_mergepairs()`

Les `bddPair*` sont des structures de données utilisées par BuDDy pour renommer des variables. Un tel renommage intervient dans Spot lorsque l'état suivant d'une transition devient l'état courant : les variables Next$[f]$ sont renommées en Now$[f]$.

Ces `bddPair*` sont des tableaux indicés par les numéros de variables, et indiquant pour chaque variable *source* le numéro de sa variable *destination*. Comme ces tableaux sont dimensionnés par rapport au nombre de variables BDD gérées par BuDDy, et que ce dernier peut évoluer au cours du temps, BuDDy doit maintenir une liste de tous ces tableaux afin de les redimensionner chaque fois que de nouvelles variables sont déclarées. D'autre part, chacun de ces tableaux possède une identité, utilisée par le cache d'opérations de BuDDy. Tout ceci empêche une manipulation extérieure de ces structures de données, et justifie l'ajout de nouvelle fonctions *dans* BuDDy.

BuDDy fournit le moyen de créer, remplir, et libérer de tels tableaux, mais ne permettait pas de les copier, ou de fusionner deux tableaux faisant des renommages disjoints. C'est ce que réalisent les fonctions `bdd_copypairs()` et `bdd_mergepairs()`.

La version actuelle de Spot n'utilise plus ces deux fonctions. Elles furent nécessaires lorsque chaque automate embarquait son propre dictionnaire de variables, avec son propre `bddPair*` pour renommer les Next$[f]$ en Now$[f]$. `bdd_copypairs` était utile lors de la copie d'automate, et `bdd_mergepairs` lors de la fusion des dictionnaires de deux automates (à l'occasion d'un produit). Aujourd'hui les automates partagent un dictionnaire et un `bddPair*` commun, et ceux-ci ne sont jamais ni copiés ni fusionnés.

`bdd_existcomp()`

`bdd_forallcomp()`

`bdd_uniquecomp()`

`bdd_appexcomp()`

`bdd_appallcomp()`

`bdd_appunicomp()`

BuDDy, propose trois type de quantification de formules propositionnelles.

La quantification existentielle d'une formule f par la variable v est la formule définie par $\exists v.f = f_{|v=1} \vee f_{|v=0}$. Pour chaque affectation de variable satisfaisant la formule quantifiée, *il existe* (au moins) une affectation de v qui permet de compléter cette affectation pour qu'elle satisfasse f.

La quantification universelle d'une formule f par la variable v est une formule définie par $\forall v.f = f_{|v=1} \wedge f_{|v=0}$. Chaque affectation de variables satisfaisant la formule quantifiée peut être complétée par *n'importe quelle* affectation de v et satisfaire f.

La quantification unique d'une formule f par la variable v est une formule définie par $f_{|v=1} \oplus f_{|v=0}$. Pour chaque affectation de variables satisfaisant la formule quantifiée, *il existe une unique* affectation de v qui permet de compléter cette affectation pour qu'elle satisfasse f.

Ces trois définitions se généralisent à la quantification de n variables. Par exemple la quantification existentielle de n variables se définit par $\exists(v_1, v_2, \ldots, v_n).f = \exists v_1.\exists v_2 \ldots \exists v_n.f$.

BuDDy propose ces quantifications généralisées avec les fonctions `bdd_exist()`, `bdd_forall()`, et `bdd_unique()`. Ces fonctions prennent pour paramètres un BDD représentant la formule, et un autre listant les variables quantificatiées.

Ces fonctions sont très souvent appelées après une opération arithmétique entre deux formules. Par exemple, le calcul des successeurs d'un ensemble d'états fait intervenir une formule telle que

$$\exists(\text{Now}[1], \text{Now}[2], \ldots, \text{Now}[n]).(current \wedge transition)$$

C'est-à-dire un $\wedge$ suivi d'un $\exists$. En conséquence, BuDDy fournit des fonctions qui combinent ces deux opérations. Cela permet de ne parcourir la BDD récursivement qu'une seule fois, d'où un gain de temps. Ces fonctions deux-en-un sont `bdd_appex()`, `bdd_appall()`, et `bdd_appuni()`.

Lors du développement de Spot, il est arrivé plusieurs fois que l'interface de ces fonctions ne soit pas adaptée. Le problème est qu'elles prennent en paramètre une liste des variables à faire disparaître, alors que parfois seules les variables à *conserver* sont connues. Le calcul de la liste des variables à supprimer à partir de celles que l'on souhaite garder est possible, mais il fait à son tour intervenir une quantification existentielle.

Pour cette raison nous avons introduit des versions *complémentées* de chacune de ces fonctions :
– `bdd_existcomp()`,
– `bdd_forallcomp()`,
– `bdd_uniquecomp()`,
– `bdd_appexcomp()`,
– `bdd_appallcomp()`, et
– `bdd_appunicomp()`.
Ces fonctions acceptent les mêmes paramètres que leurs cousines, mais traitent les ensembles de variables comme des variables à conserver.

4.2.2 LTL

Ce module prend en charge tout ce qui a trait aux formules de logique temporelle linéaire. Trois types d'objets y sont manipulés :
– des environnements
– des formules
– des visiteurs de formules
À cela il faut ajouter la présence d'un analyseur syntaxique, pour construire une formule (l'objet) à partir de sa représentation textuelle.

L'ensemble de ce module est déclaré dans l'espace de noms `spot::ltl`.

4.2.2.1 `spot::ltl::environment`

L'`environment` est un objet qui réalise une association entre la représentation textuelle d'une proposition atomique et son objet.

Lorsque l'analyseur syntaxique traduit une formule, il le fait dans le contexte d'un `environment`. Ceci permet de faire cohabiter des systèmes qui utilisent le même nom de proposition atomique pour désigner des choses différentes : la différence est faite en fonction de l'environnement utilisé.

Un `environment` permet aussi de définir des alias de variables ou d'en interdire l'emploi. Par exemple quand l'ensemble des propositions atomiques d'un système est connu, on veut restreindre les formules de logique temporelle à ces propositions.

Tout ceci est réalisé au moyen d'une unique méthode des `environment` : la méthode `require()`.

```
virtual formula* require(const std::string& prop_str) = 0;
```

`require()` attend un nom de proposition atomique en paramètre, et retourne la formule correspondante. Notons que `require()` retourne un formule et non précisément une proposition atomique, un environnement peut ainsi être utilisé pour définir des abréviations de sous-formules. Si la valeur retournée est le pointeur nul, alors la proposition est interdite par l'environnement, c'est ainsi que l'analyseur syntaxique peut se plaindre de propositions inconnues.

C'est aux utilisateurs ou développeurs de Spot de créer une classe fille de `spot::ltl::environnement` et d'y définir le comportement de `require()`.

Spot fournit une implémentation appelée `spot::ltl::default_environment`, qui *reconnaît* toutes les propositions atomiques qui lui sont soumises. Cette implémentation convient lorsqu'aucune limitation ne porte sur l'ensemble des propositions atomiques utilisables dans les formules LTL.

`spot::ltl::default_environment` est un singleton, on ne peut donc pas le construire. Son instance est obtenue en appelant `spot::ltl::default_environment::instance()`.

4.2.2.2 `spot::ltl::parse`

```
formula* parse(const std::string& ltl_string,
               parse_error_list& error_list,
               environment& env = default_environment::instance(),
               bool debug = false);
```

Cette fonction construit un objet `spot::ltl::formula` à partir de la représentation textuelle d'une formule (`ltl_string`). Les erreurs rencontrées lors de ce processus (parenthèses manquantes, propositions atomiques ou opérateurs inconnus, etc.) sont accumulées dans l'objet `error_list`. Le paramètre `env` fournit l'environnement dans lequel les propositions atomiques doivent être cherchées.

Cette fonction utilise un analyseur syntaxique généré par GNU Bison[1], et un analyseur lexical produit par Flex. `debug`, le dernier paramètre de `parse`, permet d'activer les traces de l'analyseur syntaxique. Ceci permet de suivre le comportement de l'automate construit par Bison pour débugger une grammaire.

Voici une version simplifiée de la grammaire utilisée. Elle ne fait pas apparaître la gestion des erreurs.

⟨formula⟩ → atomic_prop | true | false | '**(**'⟨formula⟩'**)**' | ⟨unary op⟩⟨formula⟩ | ⟨formula⟩⟨binary op⟩⟨formula⟩
⟨unary op⟩ → op_F | op_G | op_X | op_Not
⟨binary op⟩ → op_And | op_Or | op_Xor | op_Implies | op_Equiv | op_U | op_R

Les ambiguïtés sont résolues en fixant les priorités des opérateurs comme suit (du plus fort au plus faible).

1. op_Not
2. op_X
3. op_F, op_G
4. op_U, op_R
5. op_Implies, op_Equiv
6. op_And
7. op_Xor
8. op_Or

Les symboles terminaux sont reconnus par l'analyseur lexical avec les règles suivantes. Les espaces servent de délimiteurs mais sont autrement ignorés.

op_Not → '**!**'
op_Or → '||' | '|' | '**+**' | '\/'
op_And → '`&&`' | '`&`' | '**.**' | '*****' | '/\'
op_Xor → '^' | '**xor**'
op_Implies → '**=>**' | '**->**'
op_Equiv → '**<=>**' | '**<->**'
op_F → '**F**' | '**<**'
op_G → '**G**' | '**[]**'
op_U → '**U**'
op_R → '**R**' | '**V**'
op_X → '**X**' | '**()**'
true → '**1**' | '**true**'
false → '**0**' | '**false**'
atomic_prop → '`[a-zA-EH-WYZ_][a-zA-Z0-9_]*`' | '`[FGX][0-9_][a-zA-Z0-9_]*`'

Les différents synonymes reconnus pour chaque opérateur ont été récoltés dans plusieurs outils existants manipulant des formules LTL (Spin, LBTT, LTL2BA, ...)

[1] En fait, une version expérimentale de Bison produisant l'analyseur syntaxique sous la forme d'une classe C++.

La définition curieuse des propositions atomiques tient au fait que nous voulons reconnaître `XFa` comme la formule `X(F(a))` et non comme une proposition atomique. Une proposition atomique ne peut donc pas commencer par le symbole d'un opérateur unaire, à moins que ce dernier de soit suivi d'un chiffre ; par exemple `X2Fa` doit être interprété comme une proposition atomique. (`X0` est aussi traité comme une proposition atomique, bien qu'une autre interprétation puisse être `X(`**`false`**`)`.

`error_list` contient une liste d'erreurs sous la forme de paires $\langle position, diagnostic \rangle$, les positions indiquent les lignes et colonnes du début et fin de chaque erreur. Ceci permet à l'utilisateur d'afficher les messages d'erreurs comme il le souhaite.

`libspot` fournit une fonction pour réaliser un tel affichage.

```
bool format_parse_errors(std::ostream& os,
                         const std::string& ltl_string,
                         parse_error_list& error_list);
```

Par exemple voici comment `format_parse_errors` affiche les diagnostics produits lors de l'analyse syntaxique de la chaîne "`a * (a + b c`".

```
>>> a * (a + b c
               ^
syntax error, unexpected ATOMIC_PROP

>>> a * (a + b c
               ^
unexpected input ignored

>>> a * (a + b c
        ^^^^^^
missing closing parenthesis
```

La fonction `parse` fait de son mieux pour construire une formule, malgré les erreurs. Dans ce cas elle retourne un objet correspondant à la formule "`a * (a + b)`".

Il existe donc deux types d'erreurs. Soit rien n'a pu être analysé et `spot::ltl::parse()` retourne `0`. Soit `parse()` a réussi à récupérer quelque chose, et cette formule est retournée. Dans tous les cas, les messages d'erreurs ont été accumulés dans l'objet `error_list`.

Il faut noter que `format_parse_errors()` peut être appelé même lorsqu'il n'y a eu aucune erreur, et que c'est d'ailleurs un moyen simple de tester l'existence d'une erreur. En effet, `format_parse_errors()` retourne **`true`** si et seulement si des diagnostiques ont été affichés.

Typiquement, le code suivant est utilisé pour lire une formule depuis une chaîne `str` :

```
spot::ltl::parse_error_list pel;
spot::ltl::formula* f = spot::ltl::parse(str, pel);

int exit_code = spot::ltl::format_parse_errors(std::cerr, str, pel);

if (f)
  {
    // do something with f
  }
else
  {
    exit_code = 1;
  }

exit(exit_code);
```

4.2.2.3 `spot::ltl::formula`

`spot::ltl::formula` est un type de donnée représentant l'arbre de syntaxe abstraite d'une formule LTL. Toutes les nœuds de cet arbre sont des d'objets dérivant de `spot::ltl::formula`. La figure 4.2 montre la hiérarchie de ces formules.

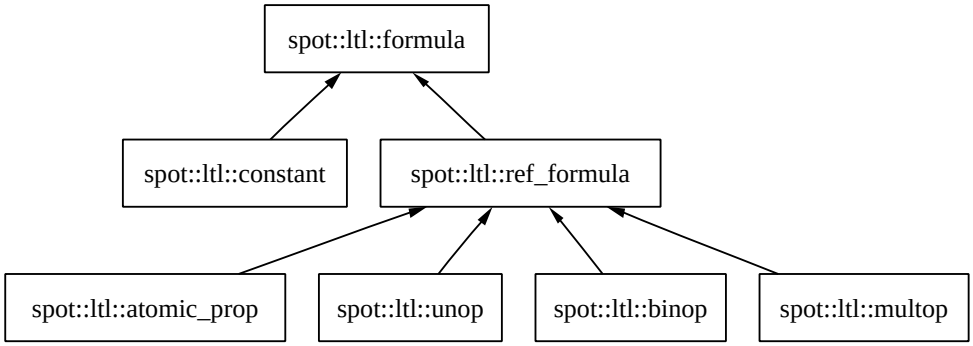
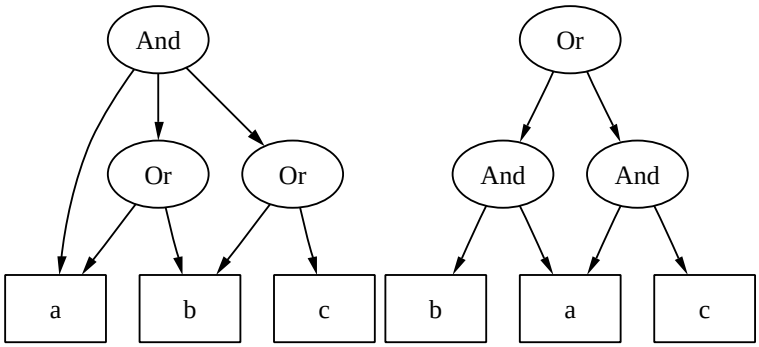


FIG. 4.2 – Hiérarchie de `spot::ltl::formula`

Les algorithmes manipulant des formules par la suite peuvent tirer parti de l'identification des sous-formules identiques. Par exemple lors de la construction d'un automate, il est intéressant de reconnaître les sous-formules identiques, pour leur affecter le même état.

Pour faciliter les comparaisons de formules, `libspot` utilise une approche similaire à celle employée dans les BDDs pour représenter ses formules. Les arbres de syntaxe abstraite sont des singletons : deux arbres identiques posséderont la même adresse. (Comme des sous-formules peuvent partager le même arbre de syntaxe abstraite, cet arbre est en fait un graphe acyclique dirigé.) Ainsi la comparaison de deux arbres peut se faire à l'aide d'une simple comparaison de pointeurs.

Il faut noter cependant que si l'égalité de deux arbres implique celle des formules qu'ils représentent, la réciproque est fausse. Comme aucune forme canonique n'est imposée, deux formules équivalentes peuvent avoir des représentations différentes. Par exemple les formules $a \wedge (a \vee b) \wedge (b \vee c)$ et $(a \wedge b) \vee (a \wedge c)$ sont logiquement équivalentes mais possèdent des arbres de syntaxe abstraite différents (figures 4.3(a) et 4.3(b) respectivement).



(a) (b)

FIG. 4.3 – Arbres de syntaxe abstraite des formules $a \wedge (a \vee b) \wedge (b \vee c)$ et $(a \wedge b) \vee (a \wedge c)$

Les différents types de nœuds peuvent effectuer quelques simplifications lors de leur constructions, par exemple la suppression des doublons parmi les fils (ainsi $a \wedge b \wedge b$ sera représenté par le même arbre que $a \wedge b$), et le tri des fils ($a \wedge b$ sera représenté par le même arbre que $b \wedge a$). Ceci permet de détecter plus d'équivalences, mais dans l'ensemble la comparaison de deux arbres de syntaxe abstraite reste une comparaison *syntaxique* et non logique.

Comme un arbre de syntaxe abstrait peut-être partagé par plusieurs formules, chaque nœud de l'arbre possède son propre compteur de références. Ce comptage est implémenté dans la classe abstraite `ref_formula`, dont dérivent tous les types de nœuds, les constantes exceptées. Les constantes **`true`** et **`false`** sont représentées par des instances de la classe `spot::ltl::constant` instanciée une fois pour toute la durée de vie de `libspot`, il est donc inutile d'en compter les références. Leurs instances s'obtiennent en appelant respectivement les méthodes de classe `spot::ltl::constant::true_instance()` et `spot::ltl::constant::false_instance()`, et il est impossible d'en construire de nouvelles.

Les autres types des nœuds ne peuvent pas non plus être construits directement afin d'en assurer l'unicité. La construction se fait implicitement en appelant une méthode de classe `instance()` paramétrée par les caractéristiques du nœud (opérateur, fils), qui créera l'objet s'il n'existe pas déjà. Les différents types de nœuds sont les suivants.

`atomic_prop`
: Une proposition atomique.

`unop`
: Une opération unaire telle que X, F, G, ou ¬[2].

`binop`
: Une opération binaire telle que U, R, ⇔, ⇒, ou ⊕.

`multop`
: Une opération n-aire, telle que ⋁ ou ⋀.

L'intérêt de ne pas définir ∧ et ∨ comme des opérations binaires est de représenter $(a \wedge b) \wedge c$ et $a \wedge (b \wedge c)$ par le même arbre de syntaxe abstraite. Cela favorise les simplifications et les réécritures. La figure 4.3(a) montre un ∧ ternaire.

Les propositions atomiques sont habituellement construites indirectement en appelant la méthode `require()` d'un environnement. Charge à ce dernier d'appeler `spot::ltl::atomic_prop::instance()`.

Afin d'illustrer les interactions entre l'analyseur syntaxique, l'environnement qui lui est passé, et les différentes sous-classes de formules, la figure 4.4 montre la séquence d'appels effectués lors de l'analyse de `"a * (b + c)"`. L'objet `yy::Parser` est l'analyseur syntaxique construit par GNU Bison, c'est lui qui réalise tout le travail de la fonction `parse` présentée section 4.2.2.2. L'objet `env` est l'environnement passé en argument à `parse`, on voit que toutes les recherches de propositions atomiques passent par lui.

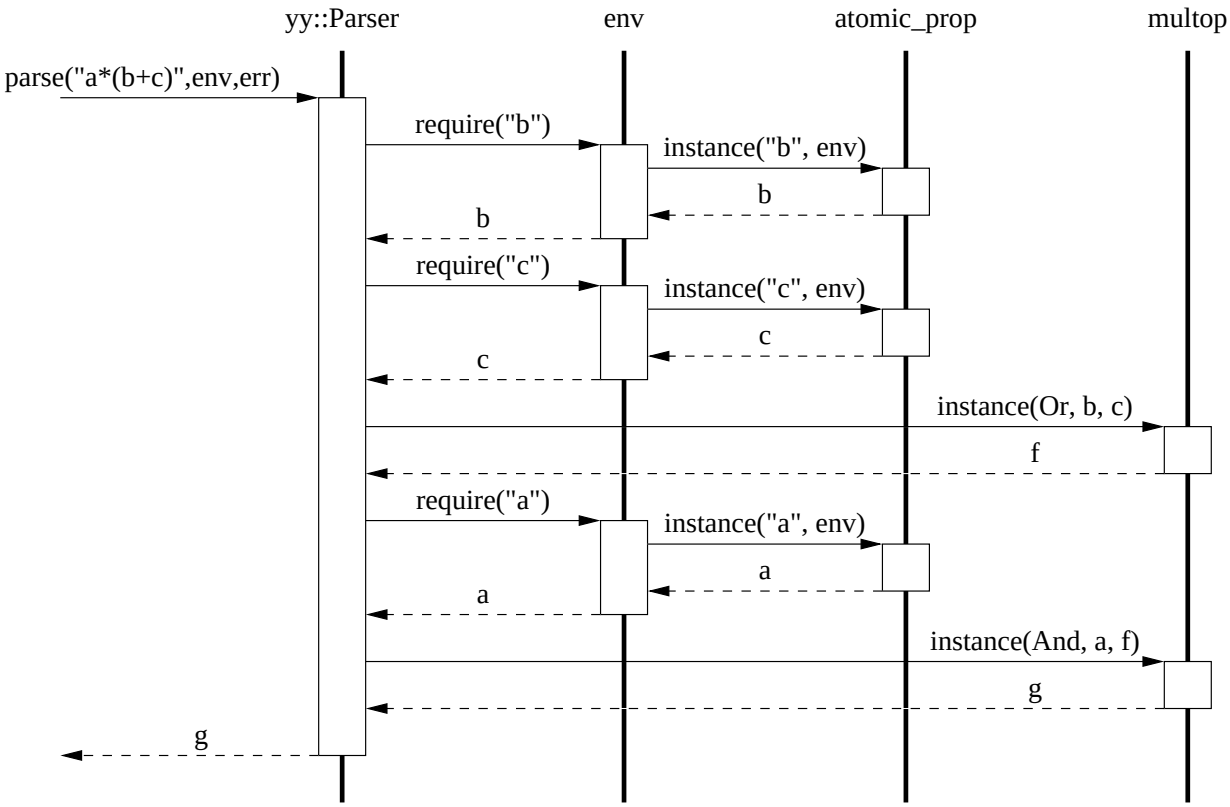


FIG. 4.4 – Diagramme de séquence de l'analyse syntaxique de la chaîne `"a * (b + c)"`

4.2.2.4 `spot::ltl::visitor` et `spot::ltl::const_visitor`

Les algorithmes manipulant des `formula` doivent constamment faire des choix en fonction du type dynamique de la formule (`constant`, `atomic_prop`, `unop`, `binop`, ou `multop`).

Dans Spot, cette sélection est effectuée par l'utilisation du patron de conception *visiteur* [27]. Un visiteur permet de réaliser l'équivalent d'un **`switch`** sur le type d'une formule, d'exécuter des actions spécifiques à ce type, sans avoir à faire aucune conversion de type.

[2] Contrairement à la figure 2.2 page 11, les négations sont représentées par des nœuds et peuvent apparaître à n'importe quel niveau de la formule. La fonction `negative_normal_form()` décrite par la suite peut être utilisée pour mettre une formule sous la forme de la figure 2.2.

Concrètement, un visiteur est une classe implémentant une méthode `visit`, pour chacun des types concrets de la hiérarchie `formula`. L'interface d'un visiteur est imposée par les classes abstraites `visitor` et `const_visitor`. Comme leur nom l'indique, les `const_visitor` sont des visiteurs qui ne modifient pas les formules visitées.[3]

```
v-1   struct visitor
v-2   {
v-3     virtual void visit(constant* node) = 0;
v-4     virtual void visit(atomic_prop* node) = 0;
v-5     virtual void visit(binop* node) = 0;
v-6     virtual void visit(unop* node) = 0;
v-7     virtual void visit(multop* node) = 0;
v-8   };
v-9
v-10  struct const_visitor
v-11  {
v-12    virtual void visit(const constant* node) = 0;
v-13    virtual void visit(const atomic_prop* node) = 0;
v-14    virtual void visit(const binop* node) = 0;
v-15    virtual void visit(const unop* node) = 0;
v-16    virtual void visit(const multop* node) = 0;
v-17  };
```

Chaque `formula` est équipée d'une méthode `accept` qui reçoit un visiteur, et y appelle la bonne méthode `visit`.

```
f-1   class formula
f-2   {
f-3   public:
f-4     // ...
f-5     virtual void accept(visitor& v) = 0;
f-6     virtual void accept(const_visitor& v) const = 0;
f-7     // ...
f-8   };
f-9
f-10  class constant : public formula
f-11  {
f-12  public:
f-13    // ...
f-14    virtual void accept(visitor& v) { v.visit(this); }
f-15    virtual void accept(const_visitor& v) const { v.visit(this); }
f-16    // ...
f-17  };
```

Ainsi lors de l'appel de `v.visit(this)`, ligne f-14, le type de la formule est connue, et c'est bien la méthode `visit(constant* node)`, ligne v-3, qui est appelée.

Tous les algorithmes manipulant des formules sont écrits sous forme de visiteurs. En fait, les méthodes offertes par les objets `formula` sont réduites à leur strict minimum. Même des opérations telles que le clonage d'une formule ou sa destruction sont implémentés par des visiteurs. Ceci permet de créer de nouveaux algorithmes en raffinant des visiteurs existants par héritage.

Par exemple le clonage d'une formule est une opération qui consiste simplement à incrémenter le compteur de référence de chaque formule.

```
class clone_visitor : public visitor
{
public:
  clone_visitor() {}
  virtual ~clone_visitor() {}

  virtual void visit(constant* c)     { result_ = c->ref(); }
  virtual void visit(atomic_prop* ap) { result_ = ap->ref(); }
  virtual void visit(unop* uo) {
```

[3] Cette différence importe peu dans la version actuelle de Spot. En effet, la seule modification possible d'une formule est celle de son compteur de référence. Comme les formules doivent posséder une adresse unique, tout autre modification, telle que le changement de l'opérateur ou de l'un des fils, ne peut se faire qu'en créant une autre formule.

```
      result_ = unop::instance(uo->op(), recurse(uo->child()));
    }
    virtual void visit(binop* bo) {
      result_ = binop::instance(bo->op(),
                                recurse(bo->first()),
                                recurse(bo->second()));
    }
    virtual void visit(multop* mo) {
      multop::vec* res = new multop::vec;
      unsigned mos = mo->size();
      for (unsigned i = 0; i < mos; ++i)
        res->push_back(recurse(mo->nth(i)));
      result_ = multop::instance(mo->op(), res);
    }

    virtual formula* recurse(formula* f) { return clone(f); }
    formula* result() const { return result_; }
  protected:
    formula* result_;
  };

  formula* clone(const formula* f)
  {
    clone_visitor v;
    const_cast<formula*>(f)->accept(v);
    return v.result();
  }
```

Cette classe illustre deux idiomes utilisés lors de l'écriture d'un visiteur : `result()` et `recurse()`.

Les méthodes `visit()` ne retournent aucune valeur, car il serait impossible de déclarer un type de retour commun à tous les visiteurs. L'approche habituelle est de stocker le résultat comme un attribut du visiteur, puis d'interroger le visiteur une fois qu'il a visité la formule. C'est ce qui est fait dans la fonction `clone()` : le résultat est obtenu en appelant `result`. Les méthodes `accept()`, appellent le visiteur sur le nœud courant. La décision de parcourir les fils de ce nœud (et dans quel sens), est laissée au visiteur. Du point de vue d'un visiteur, un parcours récursif peut se faire de deux façons. Soit le visiteur s'envoie lui-même vers les fils. Par exemple c'est ce que fait le `dump_visitor` pour afficher les formules :

```
  class dump_visitor : public const_visitor
  {
  public:
     dump_visitor(std::ostream& os = std::cout)
       : os_(os) {}
   // ...
    void visit(const binop* bo) {
      os_ << "binop(" << bo->op_name() << ", ";
      bo->first()->accept(*this);
      os_ << ", ";
      bo->second()->accept(*this);
      os_ << ")";
    }
    // ...
  private:
    std::ostream& os_;
  };
```

Soit une nouvelle instance de visiteur est envoyée au fils. Cette seconde approche est nécessaire lorsque la visite d'une formule modifie des attributs du visiteur. Parfois, les deux techniques peuvent être employées. Par exemple le `dotty_visitor`, qui affiche les formules sous forme d'un arbre au format `dot`, possède un attribut `father_` qui indique le père du nœud courant, afin de dessiner les branches de l'arbre. Ce père doit être mis à jour avant dans descendre dans chacun des fils. Cela est fait de la façon suivante :

```
1  class dotty_visitor : public const_visitor
2  {
```

```
3   public:
4     // ...
5     void
6     visit(const binop* bo)
7     {
8       draw_node(bo);
9       father_ = /* current node */;
10      dotty_visitor v(*this);
11      bo->first()->accept(v);
12      bo->second()->accept(*this);
13    }
14  private:
15    int father_;
16    // ...
17  };
```

Ici il aurait été impossible d'appeler

```
12      bo->first()->accept(*this);
13      bo->second()->accept(*this);
```

car puisque la variable `father_` est modifiée à chaque niveau de la formule, il n'a plus la même valeur après l'appel de `bo->first()->accept(*this)`. Il était donc nécessaire soit de cloner le visiteur (ligne 10), soit de sauvegarder les attributs importants.

Lorsqu'un visiteur crée de nouvelles instances, cela pose un problème lors de l'héritage. Par exemple la méthode `dotty_visitor::visit()` crée explicitement un visiteur du type `dotty_visitor`. Cela signifie que toute sous classe `X` de `dotty_visitor` qui hérite de cette méthode sans l'écraser explorera tous les fils gauches de `binop` avec un `dotty_visitor` au lieu d'un `X`.

Dans le cas de `dotty_visitor` le problème ne se pose pas car il s'agit d'une classe privée, non visible (donc non sous-classable) par l'utilisateur. Pour les autres, comme `clone_visitor`, que l'on veut pouvoir sous-classer, les opérations de création d'une instance de visiteur, et de la récursion dans un fils sont abstraites par une méthode appelée `recurse()`, qui doit être redéfinie dans les sous-classes. Dans le cas des visiteurs qui retournent une valeur, `recurse()` prend aussi en charge l'appel de `result()` sur le visiteur créé et le retour de cet valeur.

Le visiteur `unabbreviate_logic_visitor` illustre l'héritage d'algorithmes par sous-classage de visiteur. Ce visiteur récrit une formule en supprimant les opérateurs binaires tels que $\Leftrightarrow$, $\Rightarrow$, et $\oplus$, pour les récrire simplement avec $\wedge$, $\vee$, et $\neg$. On devine que seuls les nœuds de type `spot::ltl::binop` doivent être modifiés, les autres seront simplement clonés dans la formule résultante, sans autre modification. `unabbreviate_logic_visitor` peut donc être écrit comme une sous-classe de `clone_visitor` qui redéfinit les méthodes `visit(binop*)` et `recurse(formula*)`.

```
class unabbreviate_logic_visitor : public clone_visitor
{
  typedef clone_visitor super;
public:
  unabbreviate_logic_visitor() {}
  virtual ~unabbreviate_logic_visitor() {}

  using super::visit;

  virtual void visit(binop* bo) {
    formula* f1 = recurse(bo->first());
    formula* f2 = recurse(bo->second());
    switch (bo->op())
      {
      /* f1 ^ f2  ==  (f1 & !f2) | (f2 & !f1) */
    case binop::Xor:
      result_ =
        multop::instance(multop::Or,
                         multop::instance(multop::And, clone(f1),
                                          unop::instance(unop::Not,
                                                         f2)),
                         multop::instance(multop::And, clone(f2),
```

```
                                              unop::instance(unop::Not,
                                                             f1)));
        return;
        /* f1 => f2  ==  !f1 | f2 */
      case binop::Implies:
        result_ = multop::instance(multop::Or,
                                   unop::instance(unop::Not, f1), f2);
        return;
        /* f1 <=> f2  ==  (f1 & f2) | (!f1 & !f2) */
      case binop::Equiv:
        result_ =
          multop::instance(multop::Or,
                           multop::instance(multop::And,
                                            clone(f1), clone(f2)),
                           multop::instance(multop::And,
                                            unop::instance(unop::Not,
                                                           f1),
                                            unop::instance(unop::Not,
                                                           f2)));
        return;
        /* f1 U f2 == f1 U f2 */
        /* f1 R f2 == f1 R f2 */
      case binop::U:
      case binop::R:
        result_ = binop::instance(bo->op(), f1, f2);
        return;
      }
    /* Unreachable code. */
    assert(0);
  }

  virtual formula* recurse(formula* f) { return unabbreviate_logic(f); }
};

formula*
unabbreviate_logic(const formula* f)
{
  unabbreviate_logic_visitor v;
  const_cast<formula*>(f)->accept(v);
  return v.result();
}
```

Il est amusant de noter que le `clone_visitor` est utilisé de deux façons ici. D'une part il est utilisé comme classe mère, et les deux appels à `recurse` au début de `visit(binop*)` feront sans doute appel à certaines de ses méthodes (sauf si toute la formule est composée exclusivement de `binop`). D'autre part, bien que l'appel à `recurse` retourne une formule clonée, lorsque la formule apparaît deux fois dans la réécriture (comme c'est le cas pour $f_1 \Leftrightarrow f_2 = (f_1 \wedge f_2) \vee (\neg f_1 \wedge \neg f_2)$, il faut cloner la formule une seconde fois : c'est la raison des appels explicites à `clone()`.

Un tel sous-classage peut être continué pour raffiner à nouveau le comportement. Par exemple, en plus de supprimer les abréviations d'opérateurs de logique, le visiteur `unabbreviate_ltl_visitor` va aussi supprimer celles d'opérateurs de logique temporelle (G et F). Ce visiteur est une sous-classe de `unabbreviate_logic_visitor` qui redéfinit `visit(unop*)` et bien sûr `recurse(formula*)`.

À l'heure actuelle, les visiteurs suivants ont été implémentés. Tous sont utilisables à travers des fonctions (telles que `clone()`) mais certains sont aussi définis de façon publique de façon à pouvoir servir de bases à d'autres algorithmes.

`clone_visitor`
: Traverse une formule en incrémentant les compteurs de références de chaque nœud. Utilisé par la fonction `clone()`, ainsi que comme classe de base d'autres visiteurs.

`destroy_visitor`
: Visiteur interne utilisé par la fonction `destroy()`, c'est une sous-classe de `postfix_visitor`. Il s'agit

du contraire de `clone()` : il décrémente les compteurs de références de chacun des nœuds de la formule traversée.

`dotty_visitor`
Visiteur interne utilisé par la fonction `dotty()`. Parcours une formule pour la représenter sous la forme d'un graphe au format `dot`. Les figures 4.3(a) et 4.3(b) ont été produites par cette fonction.

`unabbreviate_logic_visitor`
Sous-classe de `clone_visitor` qui en plus de cloner la formule, récrit les opérateurs $\Leftrightarrow$, $\Rightarrow$, et $\oplus$ à l'aide de $\wedge$, $\vee$, et $\neg$. Utilisé par la fonction `unabbreviate_logic`.

`negative_normal_form_visitor`
Visiteur interne utilisé par la fonction `negative_normal_form()`. Récrit une formule en poussant toutes les négations vers les propositions atomiques.

`postfix_visitor`
Un visiteur qui ne fait rien, à part parcourir une formule et appeler une méthode (`doit()`) sur chacun de ses nœuds dans l'ordre postfixe. Les sous-classes de `postfix_visitor` doivent redéfinir cette méthode. Ce visiteur est publique et faciliter l'écriture d'une grande classe d'algorithmes sur les formules.

`to_string_visitor`
Un visiteur interne utilisé par la fonction `to_string()`. Il affiche une formule sous un format qui peut à nouveau être accepté par l'analyseur syntaxique.

`unabbreviate_ltl_visitor`
Sous-classe de `unabbreviate_logic_visitor`, qui en plus de récrire certains opérateurs de logique, va récrire les opérateurs de logique temporelle G et F.

Pour conclure cette section et illustrer le fonctionnement d'un visiteur, la figure 4.5 montre la traversée de la formule `g` (créée figure 4.4) par une instance de `dump_visitor()`. Cette opération affiche la chaîne

```
Binop(And, AP(a), Binop(Or, AP(b), AP(c)))
```

4.2.3 TGBA

Ce module de `libspot` gère les automates de Büchi généralisés étiquetés sur les transitions. Trois types d'objets jouent un rôle important aux côtés de ces automates : des dictionnaires qui associent une signification à chaque variable BDD, des états qui désignent une position dans un automate, et des itérateurs sur les successeurs d'un état. Nous décrivons ces trois types d'objet avant de décrire les automates et les algorithmes qui les utilisent.

Tous ces objets appartiennent à l'espace de nom `spot`.

4.2.3.1 Dictionnaires

Lorsque BuDDy est initialisé, il faut fixer le nombre de variables qui seront utilisées. Ce nombre peut être augmenté par la suite, au besoin. Du point de vue de BuDDy, les variables sont globales au programme, et sont numérotées de 0 à $varnum - 1$.

Dans `libspot`, chaque variable a une signification, par exemple elles peuvent servir à désigner une variable, un état. D'autre part les variables sont utilisées localement : une même variable peut posséder une signification différente selon l'endroit où elle est utilisée.

Les dictionnaires formalisent l'association entre les variables et leur signification. Dans la (très) courte histoire de `libspot`, les dictionnaires ont changé deux fois d'utilisation, et cela n'est pas terminé.

Dans les premières versions, chaque automate possédait son propre dictionnaire, afin d'utiliser au maximum les variables BDD existantes sans devoir en ajouter de nouvelles. La conséquence était que la même variable BDD (pour BuDDy ce n'est qu'un numéro) pouvait désigner deux propositions atomiques différentes dans deux automates. Pour réaliser le produit synchronisé de deux automates il fallait donc trouver un terrain d'entente : un troisième dictionnaire, unifiant les dictionnaires des deux automates à synchroniser devait être créé pour le produit, puis les appels à chacun des automates pendant le produit devaient être entourés de traductions entre les différents dictionnaires. Cette approche était assez fastidieuse, lente à cause des traductions, et empêchait des optimisations telles que celle basée sur `support_conditions()` qui sera décrite section 4.2.3.9.

Dans l'approche actuelle, les dictionnaires peuvent être partagés par plusieurs automates. Les automates peuvent toujours posséder leur propre dictionnaire, un produit de deux automates ne peut être réalisé qu'entre deux automates partageant le même dictionnaire. Ceci évite toute traduction, et permet l'optimisation précitée. Typiquement tous les automates partageront le même dictionnaire. La seule raison pour laquelle les automates ont gardé la possibilité d'en

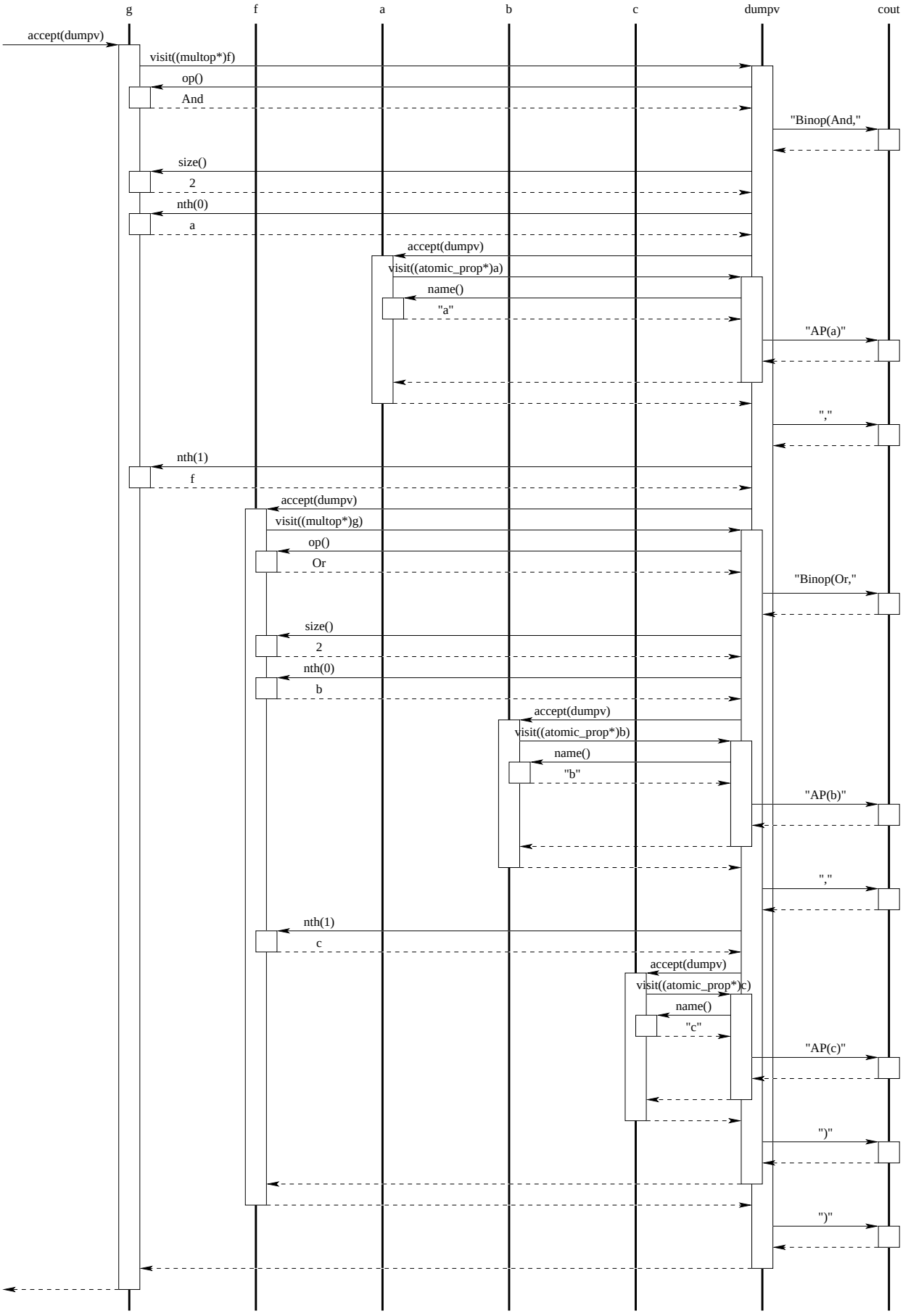


FIG. 4.5 – Diagramme de séquence de la visite de l'objet `g` par un `dump_visitor`.

utiliser un autre est que cela permet éventuellement de travailler sur deux problèmes disjoints sans devoir doubler le nombre de variables BDD.

La classe `bdd_dict` implémente un tel dictionnaire. Elle classe les variables BDD selon trois types, dans trois paires de tables différentes.

1. Les variables Now[·] et Next[·] qui désignent des états de l'automate. Les variables Now[·] sont conservées dans les tables `now_map` et `now_formula_map`. Les variables Next[·] ne sont pas conservées car dans l'implémentation actuelle, elles utilisent systématiquement le numéro de variable suivant celui de la variable Now[·] correspondante.[4]
2. L'équivalence entre les propositions atomiques et leur variable BDD, est conservée dans les tables `var_map`, et `var_formula_map`.
3. Les tables `acc_map` et `acc_formula_map` stockent les variables désignant les conditions d'acception. Si un automate possède les ensembles d'acceptation $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$, alors chacun de ces n ensembles d'acceptation sera symbolisé par une variable BDD Acc$[i]$.

Dans tous les cas `*_map` sont des tableaux associant à chaque formule son numéro de variable, tandis qu'inversement les `*_formula_map` associent à chaque numéro de variable la `formula` correspondante. Les premiers sont utilisés lors des constructions d'automate, les derniers lors de leur affichage.

Les attributs `next_to_now`, et `now_to_next` contiennent les règles de réécriture utilisées par BuDDy pour transformer l'ensemble des variables Next[·] d'une formule en variables Now[·], et vice-versa.

```
class bdd_dict: public bdd_allocator
{
public:

  bdd_dict();
  ~bdd_dict();

  typedef std::map<const ltl::formula*, int> fv_map;
  typedef std::map<int, const ltl::formula*> vf_map;

  fv_map now_map;
  vf_map now_formula_map;
  fv_map var_map;
  vf_map var_formula_map;
  fv_map acc_map;
  vf_map acc_formula_map;

  bddPair* next_to_now;
  bddPair* now_to_next;

  int register_state(const ltl::formula* f, const void* for_me);
  int register_proposition(const ltl::formula* f, const void* for_me);
  int register_accepting_variable(const ltl::formula* f,
                                  const void* for_me);

  void register_all_variables_of(const void* from_other,
                                 const void* for_me);
  void unregister_all_my_variables(const void* me);

  bool is_registered_proposition(const ltl::formula* f, const void* by_me);
  bool is_registered_state(const ltl::formula* f, const void* by_me);
  bool is_registered_accepting_variable(const ltl::formula* f,
                                        const void* by_me);

  std::ostream& dump(std::ostream& os) const;
  void assert_emptiness() const;
  // ...
}
```

[4] Ce n'est pas nécessairement une bonne chose, car cela impose un ordre relatif sur ces variables dans les BDD. La variable Next$[f]$, suit toujours immédiatement la variable Now$[f]$ dans un BDD. Cela est très visible sur la figure 4.9, par exemple. D'autre part cela rend l'allocation des variables (au niveau de la classe `bdd_allocator`) un peu plus compliquée. Il serait bon de se débarrasser de cette contrainte dans le futur.

Comme un dictionnaire peut être partagé entre plusieurs automates, il doit compter les références sur chaque variable pour savoir à tout moment quelles sont les variables qui sont utilisées, ainsi que celles qui sont libres d'être réutilisées. Par conséquent, lors des définitions de nouvelles variables (avec les méthodes `register_state()`, `register_proposition()`, ou `register_accepting_variable()` selon le type de variable), le propriétaire de la variable doit indiquer son identité avec le paramètre `for_me`. En pratique, `for_me` est souvent l'adresse de l'automate utilisant la variable.

Il est parfois utile de pouvoir désigner toutes les variables utilisées par un automate. Lorsqu'un nouvel automate utilise les mêmes variables qu'un autre (cela peut arriver lorsque l'automate est copié, ou si l'un est une façade de l'autre), il doit signaler qu'il réutilise ces mêmes variables à l'aide de la méthode `register_all_variables_of`. De façon similaire, un automate libère toutes ses variables avec un simple appel à `unregister_all_my_variables()` dans son destructeur.

Débugger les utilisations de références sur ces variables est parfois difficile. La fonction `assert_emptiness()` est automatiquement appelée par le destructeur de `bdd_dict` pour s'assurer que tous les utilisateurs de ce dictionnaire ont bien été libérés. Elle peut aussi être appelée manuellement pour aider chasser des erreurs. Lorsque le dictionnaire n'est pas vide, elle appelle `dump()` pour l'afficher.

L'utilisation des dictionnaires dans `libspot` devrait encore évoluer. Pour le moment, le dictionnaire est principalement centré autour de l'utilisation qu'en fait la classe `tgba_bdd_concrete` présentée section 4.2.3.8. En particulier l'utilisation des variables Now[·] et Next[·], qui est spécifique à l'algorithme de traduction produisant des `tgba_bdd_concrete` (cf. `ltl_to_tgba_lacim()` section 4.2.4.1). Les autres types d'automates ne font que réutiliser une sous-partie de cette interface : ils utilisent des conditions d'acceptation et des propositions atomiques, mais pas de variables Now[·] ou Next[·]. En fait, ces variables Now[·] et Next[·] n'apparaissent dans aucun des BDD utilisés dans l'interface de la classe `tgba` (section 4.2.3.5). Now[·] et Next[·] devraient être considérées comme des variables privées à chaque automate `tgba`. Il serait donc préférable de disposer d'un dictionnaire à deux niveaux
– une partie partagée entre les automates contenant les propositions atomiques et les conditions d'acceptation
– une partie locale à chaque instance d'automate complétant le dictionnaire avec des variables utilisées en interne par l'automate.

La classe `bdd_allocator`, dont dérive `bdd_dict`, est un sorte de remplacement pour ce que certaines bibliothèques de BDD appellent un *BDD manager*, mais que BuDDy ne fournit pas. Les instances de `bdd_allocator` gèrent des listes de variables BDD. Elles conservent à tout moment la liste des variables inutilisées, et s'occupent de réclamer de nouvelles lorsque la liste est vide.

```
class bdd_allocator
{
protected:
  bdd_allocator();
  int allocate_variables(int n);
  void release_variables(int base, int n);
  // ...
}
```

L'interface est assez simple. Le constructeur de la classe s'occupe entre autres choses d'initialiser la bibliothèque BuDDy lorsque ce n'est déjà fait. Comme toutes les variables BDD de `libspot` sont obtenues en interrogeant des dictionnaires, cela garantit que BuDDy sera toujours initialisé.

`allocate_variables(n)` permet d'allouer n variables BDD consécutives ; la valeur de retour est le numéro de la première variable. La méthode essaye de limiter la fragmentation de la liste de variables libres en allouant les n variables dans le plus petit bloc d'au moins n variables libres.

`release_variables(base, n)` est l'opération inverse qui permet de libérer les variables numérotées de $base$ à $base + n - 1$.

4.2.3.2 États

Un *model checker* efficace chercherait à représenter les états du système à tester de la façon la plus compacte possible, afin de limiter l'espace mémoire employé (et donc pouvoir traiter de plus gros systèmes plus rapidement). Ici nous aimerions limiter l'espace occupé par les états de tous les automates. Malheureusement cela est difficilement compatible avec la volonté de s'interfacer avec des bibliothèque externes. En effet chaque bibliothèque va représenter ses états de façon différente.

Le choix effectué ici est d'utiliser une classe abstraite. Un état de Spot est une instance de la classe `state`.

```
class state
{
```

```
public:
  virtual int compare(const state* other) const = 0;
  virtual size_t hash() const = 0;
  virtual state* clone() const = 0;
};
```

La fonction `clone()` permet de dupliquer un état ; `compare()` permet de comparer deux états (avec la même convention que `strcmp()`) ; et `hash()` doit retourner un entier pouvant servir de clef dans une table de hachage.

Un état en soi n'a aucune signification s'il n'est pas rattaché à un automate. Naturellement, il est hors de question d'encombrer chaque état par un pointeur sur l'automate dont il est tiré, ce n'est normalement pas un problème pour le programmeur de savoir quel état appartient à quel automate. Mais cela justifie la pauvreté de l'interface des états. La plupart des opérations manipulant des états sont des méthodes de l'automate. Par exemple un état ne sait pas s'afficher, mais un automate sait afficher l'un de ses états.

La raison pour laquelle `compare()` et `hash()` sont des méthodes de `state`, plutôt que des méthodes des automates (on aurait très bien pu imaginer fournir une classe `state` sans aucune méthode !), est que ces deux fonctions sont utilisées dans les conteneurs de la STL [5], alors que l'automate correspondant est inconnu du conteneur. Les foncteurs suivants sont définis par `libspot`.

```
struct state_ptr_less_than
{
  bool
  operator()(const state* left, const state* right) const
  {
    assert(left);
    return left->compare(right) < 0;
  }
};

struct state_ptr_equal
{
  bool
  operator()(const state* left, const state* right) const
  {
    assert(left);
    return 0 == left->compare(right);
  }
};

struct state_ptr_hash
{
  size_t
  operator()(const state* that) const
  {
    assert(that);
    return that->hash();
  }
};
```

Cela permet de déclarer des tableaux associatifs avec `std::map` :

```
// Remember how many times each state has been visited.
typedef std::map<const spot::state*, int,
                 spot::state_ptr_less_than> seen_map;
seen_map seen;
```

ou bien sous forme de tables de hachage en utilisant les extensions à STL de l'implémentation de SGI :

```
// Remember how many times each state has been visited.
typedef Sgi::hash_map<const spot::state*, int,
                      spot::state_ptr_hash,
                      spot::state_ptr_equal> seen_map;
seen_map seen;
```

[5] Standard Template Library : la bibliothèque standard de conteneurs génériques du C++ [61, 58].

Cependant, il faut prendre garde que contrairement aux `formula`, deux `state` identiques au sens de `compare()` ne sont pas nécessairement représentés par des pointeurs vers le même objet. Cela est particulièrement important lorsqu'on manipule un tableau tel que `seen` ci-dessus. Les états qui apparaissent comme clef dans le tableau ne doivent pas être libérés avant la fin de l'utilisation du tableau, tandis que les états qui leur sont égaux, mais n'apparaissent pas directement dans le tableau peuvent être libérés au fur et à mesure. Par exemple, les opérations qui consistent à incrémenter le compteur d'un état et à libérer cet état pourraient être réalisées avec la fonction suivante.

```
void
inc_and_free(seen_map& seen, spot::state* s)
{
  seen_map::iterator i = seen.find(state);
  if (i != seen.end())
    {
      // The state already exists.
      // Delete it if it isn't used as a key.
      if (i->first != state)
        delete state;
      ++i->second;
    }
  else
    {
      seen[s] = 1;
    }
}
```

4.2.3.3 Itérateurs sur les successeurs d'un état

Les `tgba_succ_iterator` sont des objets qui fournissent la liste des transitions sortantes (et celles des états atteints correspondants) d'un état dans un automate. Un tel objet est construit par l'automate pour un état donné.

L'itérateur est sans doute le plus connu des patrons de conceptions [27]. Il permet d'accéder aux éléments d'un conteneur un à un, sans révéler la façon dont ces éléments sont stockés par le conteneur. Dans le cadre de `libspot`, où les automates ne stockent pas forcément transitions et états explicitement, ces itérateurs permettent de calculer les successeurs à la volée, au fur et à mesure des besoins de l'algorithme utilisant ces itérateurs.

L'interface d'un itérateur est la suivante.

```
class tgba_succ_iterator
{
public:
  virtual void first() = 0;
  virtual void next() = 0;
  virtual bool done() const = 0;

  virtual state* current_state() const = 0;
  virtual bdd current_condition() const = 0;
  virtual bdd current_accepting_conditions() const = 0;
};
```

Les trois premières méthodes permettent de contrôler l'itération. Un parcours de tous les successeurs d'un état prendra typiquement la forme d'une boucle comme la suivante.

```
tgba_succ_iter* s = /* ... */;
for (s->first(); !s->done(); s->next())
  {
    // ...
  }
delete s;
```

La trois autres méthodes permettent d'interroger les propriétés de la transition courante. `current_state()` indique l'état atteint, `current_condition()`, la formule de propositions atomiques étiquetant la transition, et `current_accepting_condition()`, les ensembles de conditions d'acceptation auxquels appartient la transition.

Il faut noter qu'un `tgba_succ_iterator` ne retient pas l'état source commun à toutes les transitions sur lesquelles il itère. Il n'est pas évident que ce choix ait été le bon. Plusieurs algorithmes manipulant des piles ou listes d'itérateurs (par exemple à l'occasion d'un parcours en profondeur) doivent stocker l'état correspondant à l'itérateur séparément, et cela n'est pas très commode. D'un autre côté, d'autres algorithmes n'ont pas besoin de conserver cet état ; et dans l'éventualité où la définition d'un TGBA devrait être changée pour accepter plusieurs états initiaux, un `tgba_succ_iterator` dans sa définition actuelle, pourrait être utilisé pour itérer sur les états initiaux.

4.2.3.4 Encodage des conditions d'acceptation

Les ensembles d'acceptation auxquels appartiennent les transitions d'un automate sont encodés par des formule BDD un peu spéciales.

Si un automate possède trois ensembles d'acceptation A_f, A_g, et A_h, trois variables BDD sont créées. Pratiquement, ces ensembles sont toujours associés à des formules (ici f, g, et h), donc par la suite nous désignerons ces trois variables BDD par $\mathrm{Acc}[f]$, $\mathrm{Acc}[g]$, et $\mathrm{Acc}[h]$.

Afin de pouvoir effectuer des opérations ensemblistes sur les conditions d'acceptation qui étiquettent les transitions, l'appartenance à l'ensemble A_f est représenté par la formule $\mathrm{Acc}[f] \wedge \neg\mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h]$. L'appartenance aux ensembles A_g et A_h est indiquée par $(\neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h]) \vee (\neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h])$.

4.2.3.5 Automates

Tous les automates de `libspot` sont implémentés dans des classes qui dérivent de `spot::tgba`. Cette classe abstraite définit les méthodes communes à tous les automates.

```
class tgba
{
public:
  virtual state* get_init_state() const = 0;

  virtual tgba_succ_iterator*
  succ_iter(const state* local_state,
            const state* global_state = 0,
            const tgba* global_automaton = 0) const = 0;

  virtual bdd all_accepting_conditions() const = 0;
  virtual bdd neg_accepting_conditions() const = 0;

  virtual bdd_dict* get_dict() const = 0;

  virtual std::string format_state(const state* state) const = 0;
  virtual state* project_state(const state* s, const tgba* t) const;

  bdd support_conditions(const state* state) const;
  bdd support_variables(const state* state) const;
protected:
  virtual bdd compute_support_conditions(const state* state) const = 0;
  virtual bdd compute_support_variables(const state* state) const = 0;
private:
  // ...
};
```

Les deux premières méthodes constituent l'unique moyen de « découvrir » un automate. `get_init_state()` retourne l'état initial, et `succ_iter(s)` retourne un itérateur sur les successeurs de `s` ainsi que décrit section 4.2.3.3. Les deuxième et troisième arguments de `succ_iter()` seront décrits section 4.2.3.9, ils peuvent être ignorés par les implémentations de `tgba` mais permettent autrement de réaliser certaines optimisations lors de produits synchronisés.

Les itérateurs retournés par `succ_iter()` renseignent sur l'appartenance de chaque transition à un ensemble d'acceptation (i.e., les conditions d'acceptation). Un chemin dans un automate est accepté s'il passe infiniment souvent par toutes les conditions d'acceptation de l'automate. La méthode `all_accepting_conditions()` retourne cet ensemble de conditions d'acceptation.

La méthode `neg_accepting_conditions()` retourne la conjonction des négations des variables utilisées pour représenter les conditions d'acceptation d'un automate. Son utilité sera justifiée section 4.2.3.9.

`get_dict()` retourne le dictionnaire utilisé par l'automate.

`format_state()` construit une représentation textuelle d'un état. Cette représentation est utilisée lorsque l'automate est affiché.

Lorsqu'un automate est en fait un produit de plusieurs automates, ses états sont composés des sous-états locaux à chaque automate du produit. `project_state()` permet de projeter un état sur un (sous-)automate. Nous reviendrons sur cette méthode dans la section 4.2.3.9.

`support_conditions()` et `support_variables()` calculent (lorsque cela est possible) respectivement l'ensemble des conditions sous lesquelles les transitions sortantes d'un état sont franchissables, et l'ensemble des variables qui interviennent dans les conditions de toutes les transitions sortantes. Pour des raisons d'efficacité que nous préciserons section 4.2.3.9, les appels à `support_conditions()` et `support_variables()` sont cachés de façon à ce qu'un appel identique au précédant appel retourne rapidement le même résultat. La logique de ce cache est implémentée par les méthodes `support_conditions()` et `support_variables()` à même la classe `tgba`. Lorsqu'un calcul doit être effectué (*cache miss*), ces méthodes appellent `compute_support_conditions()` et `compute_support_variables()`. Ce sont ces dernières méthodes qui doivent être implémentées dans les sous-classes. Ceci peut être vu comme une application du patron de conception *méthode patron* [27].

4.2.3.6 Hiérarchie d'automates

La figure 4.6 présente les sous-classes de `tgba` implémentées dans `spot`.

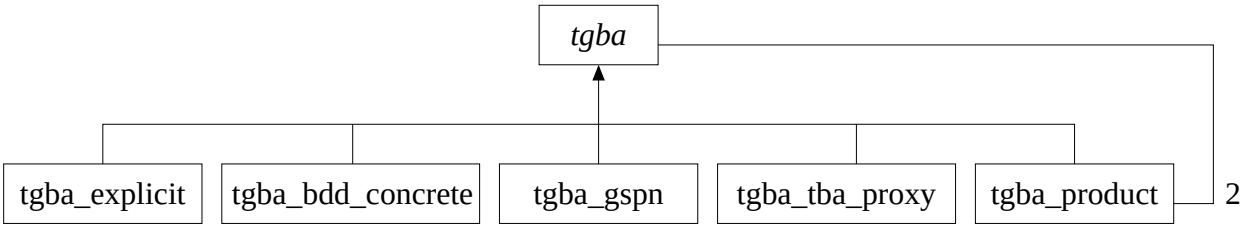


FIG. 4.6 – Hiérarchie de `tgba`

`tgba_explicit` (section 4.2.3.7)
Représente un automate en encodant ses états et transitions explicitement sous la forme d'un graphe.

`tgba_bdd_concrete` (section 4.2.3.8)
Encode un automate à l'aide de deux BDD :
– une relation de transition, qui contraint le passage d'un état à un autre, et précise les conditions à vérifier ;
– une fonction encodant les conditions d'acceptation.

`tgba_gspn` (section 4.2.5)
Présente un modèle exploré par GreatSPN comme un `tgba`.

`tgba_tba_proxy` (section 4.2.3.10)
Dégénéralise un automate à la volée. L'automate obtenu est toujours un `tgba`, mais il n'utilise qu'un seul ensemble d'acceptation.

`tgba_product` (section 4.2.3.9)
Effectue un produit de deux automates, à la volée.

4.2.3.7 `tgba_explicit`

La classe `tgba_explicit` encode un automate dans lequel les états et transitions sont stockés explicitement sous la forme d'un graphe.

La classe fut introduite à l'origine afin de pouvoir créer des automates d'exemple à la main pour tester la bibliothèque. De tels automates sont enregistrés dans un fichier texte, et lus par un analyseur syntaxique tout comme les formules LTL. Cette classe a été écrite en même temps que l'analyseur syntaxique, principalement de façon à répondre à ses besoins. Ce point justifie plusieurs choix de l'interface, par exemple le fait que les états soient désignés par un nom, ou des méthodes telles que `has_accepting_conditions()` qui permettent à l'analyseur syntaxique de diagnostiquer l'utilisation de conditions d'acceptation non déclarées.

```
class tgba_explicit: public tgba
{
public:
  tgba_explicit(bdd_dict* dict);

  struct transition;
  typedef std::list<transition*> state;
  struct transition
  {
    bdd condition;
    bdd accepting_conditions;
    state* dest;
  };

  void set_init_state(const std::string& state);

  transition*
  create_transition(const std::string& source, const std::string& dest);

  void add_condition(transition* t, ltl::formula* f);
  void add_neg_condition(transition* t, ltl::formula* f);
  void declare_accepting_condition(ltl::formula* f);
  bool has_accepting_condition(ltl::formula* f) const;
  void add_accepting_condition(transition* t, ltl::formula* f);

  // tgba interface
  virtual ~tgba_explicit();
  virtual spot::state* get_init_state() const;
  virtual tgba_succ_iterator*
  succ_iter(const spot::state* local_state,
            const spot::state* global_state = 0,
            const tgba* global_automaton = 0) const;
  virtual bdd_dict* get_dict() const;
  virtual std::string format_state(const spot::state* state) const;

  virtual bdd all_accepting_conditions() const;
  virtual bdd neg_accepting_conditions() const;

protected:
  virtual bdd compute_support_conditions(const spot::state* state) const;
  virtual bdd compute_support_variables(const spot::state* state) const;
  // ...
};
```

Pour le moment, la classe `tgba_explicit` ne supporte que des conjonctions de formules sur les transitions. Ce n'est pas un problème vis-à-vis de la hiérarchie de `tgba`, car si l'interface d'un `tgba` supporte des fonctions propositionnelles sur les transitions, elle supporte *a fortiori* les conjonctions. Mais c'est malgré tout un problème qu'il faudra régler, car puisque ce type d'automate est utilisé pour la sauvegarde en texte (cf. `save_reachable()` section 4.2.4.5) et le chargement, tous les types d'automate ne peuvent être sauvegardés.[6]

Comme il est raisonnable de l'imaginer, la méthode `set_init_state()` définit (et crée au besoin) l'état initial de l'automate. La méthode `create_transition()` permet de créer une transition ainsi que les états qu'elle relie. Cette dernière retourne un pointeur sur une transition, sur laquelle des propositions atomiques[7] peuvent être ajoutées avec les méthodes `add_condition()`, `add_neg_condition()`, et des conditions d'acceptation avec `add_accepting_condition()`.

[6] Les cas d'automates utilisant plus que des conjonctions sur leur transitions sont cependant assez rares en pratique.

[7] Le type des arguments de ces fonctions, `formula*` au lieu de simplement des `atomic_prop*`, traduit l'espoir de supporter autre chose que des conjonctions de propositions atomiques dans le futur.

4.2.3.8 `tgba_bdd_concrete`

La figure 4.7(a) représente un automate pour la formule $\mathsf{X}\, a \wedge (b \,\mathsf{U}\, \neg a)$. Cet automate a été traduit depuis la formule LTL par la fonction `ltl_to_tgba_lacim()`, décrite section 4.2.4.1 implémentant l'algorithme Couvreur/LaCIM section 2.3.1.9, puis dessiné par la fonction `dotty_reachable()` présentée section 4.2.4.5.

Mis à part l'état initial, les états sont étiquetés par des variables telles que Now$[a]$ ou Now$[b \,\mathsf{U}\, \neg a]$. Bien qu'en pratique ces variables ne soient utilisées que pour désigner les états de façon unique et doivent être regardées comme de simples identifiants, il est bon de savoir qu'elles indiquent les sous-formules qui seront vérifiées (lorsque la variable est positive) ou non (lorsqu'elle est négative) lors du franchissement des transitions sortantes.

Si un `tgba_bdd_concrete` utilise n variables Now$[\cdot]$, il peut avoir au plus $2^n + 1$ états, c'est-à-dire toutes les affectations possibles de ces variables plus l'état initial. L'état initial est un état particulier qui peut aussi faire apparaître des propositions atomiques ou des variables Next$[\cdot]$.

Chaque variable Now$[f]$ est associée à une variable Next$[f]$. Next$[f]$ signifie que Now$[f]$ sera vrai à l'état suivant. La relation de transition s'encode sous la formule d'une formule liant les variables Now$[\cdot]$ aux variables Next$[\cdot]$ et à des propositions atomiques. Par exemple, la formule Now$[\mathsf{G}\, a] \wedge a \wedge$ Next$[\mathsf{G}\, a]$ décrit un état qui boucle sur lui même en vérifiant la proposition a à chaque étape (figure 4.8).

La relation de transition créée par `ltl_to_tgba_lacim()` pour la formule $\mathsf{X}\, a \wedge (b \,\mathsf{U}\, \neg a)$ est

$$\begin{aligned} &\text{Now}[a] \Leftrightarrow a \\ \wedge &\text{Now}[b \,\mathsf{U}\, \neg a] \Leftrightarrow (\neg a \vee (b \wedge \text{Next}[b \,\mathsf{U}\, \neg a])) \end{aligned} \tag{4.1}$$

Le BDD de la formule 4.1 est donné figure 4.7(c).

De la même façon, les conditions d'acceptation sont encodées par une formule qui lie les variables Now$[\cdot]$, les propositions atomiques, et les variables Acc$[\cdot]$. La relation d'acceptation de l'automate de la figure 4.7(a) est donnée par la formule

$$(\neg\text{Now}[b \,\mathsf{U}\, \neg a] \vee \neg a) \wedge \text{Acc}[\neg a] \tag{4.2}$$

Son BDD est présenté figure 4.7(b). Cette formule signifie que toutes les transitions qui vérifient $\neg a$ ou qui proviennent d'un état étiqueté par $\neg$Now$[b \,\mathsf{U}\, \neg a]$ appartiennent à l'ensemble d'acceptation associé à $\neg a$.

La façon de calculer les successeurs d'un état est la suivante. Notons rel la relation de transition (par exemple la formule 4.1), acc la relation d'acceptation (par exemple la formule 4.2), et cur l'état courant. cur est une conjonction de variables Now$[\cdot]$, l'état initial peut aussi faire intervenir des propositions atomiques ou des variables Next$[\cdot]$. La formule suivante représentent toutes les transitions sortantes de cur.

$$cur \wedge rel \tag{4.3}$$

Si l'on considère l'ensemble des satisfactions de la formule 4.3 qui affectent une valeur à chaque variable Next$[\cdot]$, on obtient l'ensemble des successeurs. Par exemple utilisons l'équation 4.1 comme relation de transition, et choisissons de calculer les successeurs de $cur =$ Now$[a] \wedge \neg$Now$[b \,\mathsf{U}\, \neg a]$. On a

$$\begin{aligned} cur \wedge rel =& \text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a] \\ & \wedge \text{Now}[a] \Leftrightarrow a \\ & \wedge \text{Now}[b \,\mathsf{U}\, \neg a] \Leftrightarrow (\neg a \vee (b \wedge \text{Next}[b \,\mathsf{U}\, \neg a])) \\ =& \text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a] \\ & \wedge a \\ & \wedge \neg(\neg a \vee (b \wedge \text{Next}[b \,\mathsf{U}\, \neg a])) \\ =& \text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a] \\ & \wedge a \\ & \wedge (a \wedge (\neg b \vee \neg\text{Next}[b \,\mathsf{U}\, \neg a])) \\ =& (cur \wedge a \wedge \neg b) \\ & \vee (cur \wedge a \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a])) \end{aligned} \tag{4.4}$$

Cette équation fait apparaître deux types de successeurs. Les transitions qui vérifient $a \wedge \neg b$ peuvent être connectées à n'importe quel état, car il n'y a aucune contrainte sur les variables Next$[\cdot]$. Celles qui ne vérifient que a, doivent

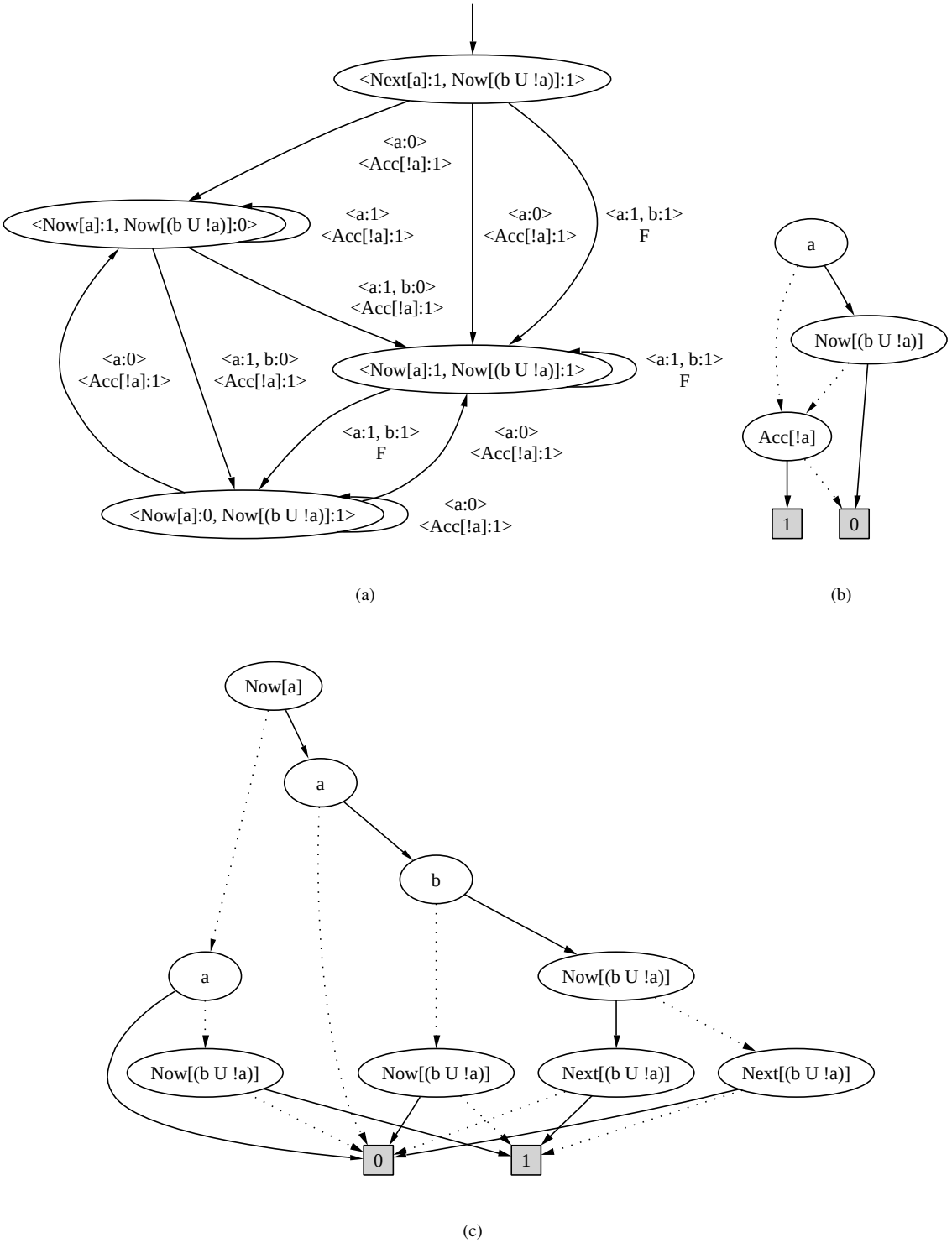


FIG. 4.7 – (a) Automate reconnaissant $\mathsf{X}\, a \wedge (b \,\mathsf{U}\, \neg a)$. (b) Encodage des conditions d'acceptation. (c) Encodage de la relation de transition.

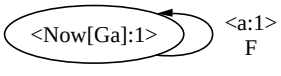


FIG. 4.8 – Transition représentée par la formule $\text{Now}[\mathsf{G}\, a] \wedge a \wedge \text{Next}[\mathsf{G}\, a]$.

nécessairement être connectées à des états vérifiant $\neg\text{Now}[b \mathsf{U} \neg a]$. Développons pour faire apparaître toutes les combinaisons de Next[.] possibles.

$$\begin{aligned} cur \wedge rel = &(cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \mathsf{U} \neg a]) \\ &\vee (cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \neg\text{Next}[b \mathsf{U} \neg a]) \\ &\vee (cur \wedge a \wedge \neg b \wedge \neg\text{Next}[a] \wedge \text{Next}[b \mathsf{U} \neg a]) \\ &\vee (cur \wedge a \wedge \neg b \wedge \neg\text{Next}[a] \wedge \neg\text{Next}[b \mathsf{U} \neg a]) \\ &\vee (cur \wedge a \wedge \text{Next}[a] \wedge \neg\text{Next}[b \mathsf{U} \neg a])) \\ &\vee (cur \wedge a \wedge \neg\text{Next}[a] \wedge \neg\text{Next}[b \mathsf{U} \neg a]) \end{aligned} \tag{4.5}$$

L'équation 4.5 décrit six transitions, atteignant quatre états différents. L'état destination d'une transition s'obtient en prenant une conjonction, par exemple $t = cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \mathsf{U} \neg a]$, en ignorant toutes les variables qui ne sont pas du type Next[·] (cela se fait par quantification existentielle), et en remplaçant les variables Next[·] par les variables Now[·] correspondantes.

$$(\exists(\text{Now}[a], \text{Now}[b \mathsf{U} \neg a], a, b).t)[\text{Now}[\cdot]/\text{Next}[\cdot]] = \text{Now}[a] \wedge \text{Now}[b \mathsf{U} \neg a] \tag{4.6}$$

Ainsi t a pour destination l'état $\text{Now}[a] \wedge \text{Now}[b \mathsf{U} \neg a]$.

Certains états apparaissant comme destination dans l'équation 4.5 ne vérifient pas la contrainte imposée par rel. C'est le cas de $\neg\text{Now}[a] \wedge \neg\text{Now}[b \mathsf{U} \neg a])$, en effet,

$$\neg\text{Now}[a] \wedge \neg\text{Now}[b \mathsf{U} \neg a] \wedge rel = a \wedge \neg a = \bot$$

De tels états n'ont aucun successeur et n'ont donc aucune utilité du point de vue de l'automate (ils ne permettent de reconnaître aucun mot). Il est possible d'empêcher l'exploration de tels états en modifiant la relation de transition rel pour que les contraintes entre les variables de Now[·] soient aussi appliquées aux variables Next[·]. Par exemple, en notant AP l'ensemble des propositions atomiques :

$$rel' = rel \wedge (\exists(\text{Next}[\cdot], AP).rel)[\text{Next}[\cdot]/\text{Now}[\cdot]] \tag{4.7}$$

La figure 4.9 montre le BDD représentant la relation de transition contrainte selon l'équation 4.7 pour l'automate 4.7(a).

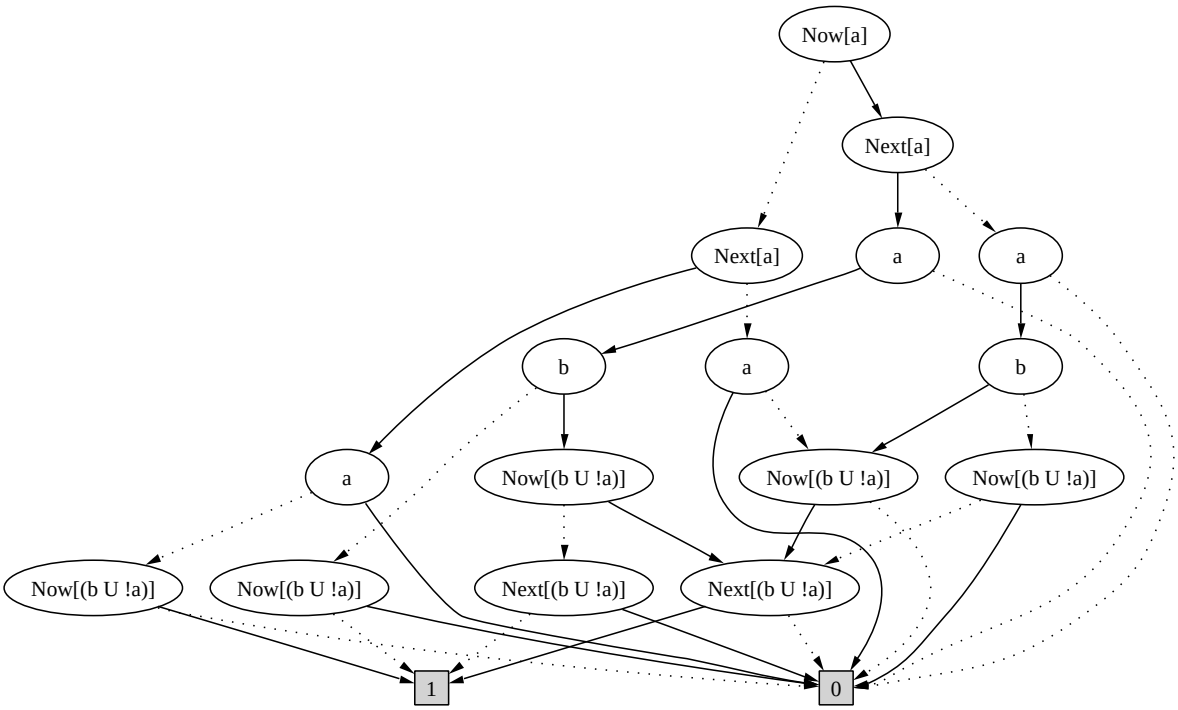


FIG. 4.9 – Relation de transition contrainte pour l'automate de la figure 4.7(a).

Avec une telle relation le calcul des successeurs ne fait plus apparaître que quatre termes.

$$\begin{aligned} cur \wedge rel' =& (cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \,\mathsf{U}\, \neg a]) \\ & \vee (cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a]) \\ & \vee (cur \wedge a \wedge \neg b \wedge \neg\text{Next}[a] \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a]) \\ & \vee (cur \wedge a \wedge \text{Next}[a] \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a])) \end{aligned} \tag{4.8}$$

Mais de ces quatre termes, le second est impliqué par le dernier. De telles simplifications sont effectuées automatiquement par les BDDs.

$$\begin{aligned} cur \wedge rel' =& (cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \,\mathsf{U}\, \neg a]) \\ & \vee (cur \wedge a \wedge \neg b \wedge \neg\text{Next}[a] \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a]) \\ & \vee (cur \wedge a \wedge \text{Next}[a] \wedge \neg\text{Next}[b \,\mathsf{U}\, \neg a])) \end{aligned} \tag{4.9}$$

Nous ne devons donc considérer que trois transitions. Ce sont les trois transitions qui apparaissent en sortie de l'état $\text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a]$ figure 4.7(a).

Le calcul des conditions d'acceptation associées à chacune de ces transitions se fait en composant la conjonction de variables désignant la transition avec la relation d'acceptation, puis en supprimant toutes les variables qui ne sont pas du type $\text{Acc}[\cdot]$. Par exemple, en utilisant la relation d'acceptation de l'équation 4.2 :

$$\begin{aligned} &\exists(\text{Now}[\cdot], \text{Next}[\cdot], a, b).(t \wedge acc) \\ &\quad =\exists(\text{Now}[\cdot], \text{Next}[\cdot], a, b).(\text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a] \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \,\mathsf{U}\, \neg a] \\ &\qquad\qquad \wedge (\neg\text{Now}[b \,\mathsf{U}\, \neg a] \vee \neg a) \wedge \text{Acc}[\neg a])) \\ &\quad =\exists(\text{Now}[\cdot], \text{Next}[\cdot], a, b).(cur \wedge a \wedge \neg b \wedge \text{Next}[a] \wedge \text{Next}[b \,\mathsf{U}\, \neg a] \wedge \text{Acc}[\neg a]) \\ &\quad =\text{Acc}[\neg a] \end{aligned} \tag{4.10}$$

D'après les équations 4.6 et 4.10, la transition t à destination de $\text{Now}[a] \wedge \text{Now}[b \,\mathsf{U}\, \neg a]$ appartient à l'ensemble d'acceptation $\text{Acc}[\neg a]$. D'autre part, il aisé de calculer les propositions atomiques qu'elle doit vérifier en supprimant toutes les autres variables :

$$\exists(\text{Now}[\cdot], \text{Next}[\cdot]).t = a \wedge \neg b \tag{4.11}$$

En pratique le calcul des successeurs est un peu plus compliqué[8] que ne le laisse penser l'équation 4.9. En effet, nos `tgba` peuvent être étiquetés par des formules de propositions atomiques, pas simplement des conjonctions, donc nous allons chercher à regrouper en une seule transition tous les arcs qui ont la même destination et les mêmes ensembles d'acceptation.

Au niveau de l'implémentation, la classe `tgba_bdd_concrete` représente un automate stocké de cette façon. Elle produit des états de type `state_bdd`, qui stockent des états tels que $\text{Now}[a] \wedge \neg\text{Now}[b \,\mathsf{U}\, \neg a]$, et des itérateurs de type `tgba_succ_iterator_concrete` qui effectuent le travail décrit par les équations 4.9 (séparation des successeurs), et 4.6, 4.10, et 4.11 (calcul des attributs de chaque transition). Le calcul de $cur \wedge rel'$ est fait par l'automate (la fonction `tgba_bdd_concrete::succ_iter()` est décrite page 72).

Les relations rel et acc sont stockées par l'automate dans un objet de type `tgba_bdd_core_data` qui contient aussi plusieurs ensembles de variables utiles lors des quantifications de BDD. Enfin, l'automate contient un pointeur vers son dictionnaire. Les liens entre toutes ces classes sont représentés figure 4.10.

4.2.3.9 `tgba_product`

La classe `tgba_product` effectue le produit synchronisé de deux automates, à la volée. Chaque instance de `tgba_product` contient deux pointeurs vers les `tgba` à utiliser comme opérandes gauche et droit du produit. Lorsque `tgba_product` crée un état (`state_product`) ou un itérateur (`tgba_succ_iterator_product`), il fournit à chacun les pointeurs vers les deux états ou itérateurs correspondant dans les opérandes. La figure 4.11 montre les relations entre toutes ces classes.

Le produit en lui-même est réalisé par les instances `tgba_succ_iterator_product`. Lorsque la méthode `tgba_product::succ_iter()` est appelée, elle construit un `tgba_succ_iterator_product` à partir

[8] La fonction qui effectue ce calcul a été récrite cinq fois en l'espace de trois mois.

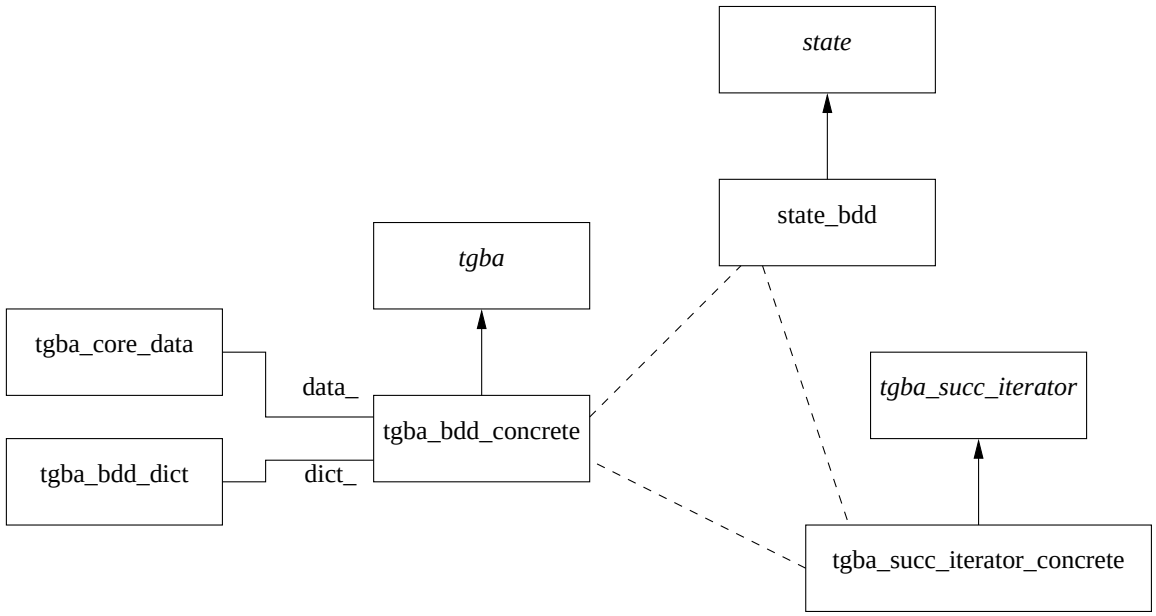


FIG. 4.10 – `tgba_bdd_concrete` et classes collaboratrices

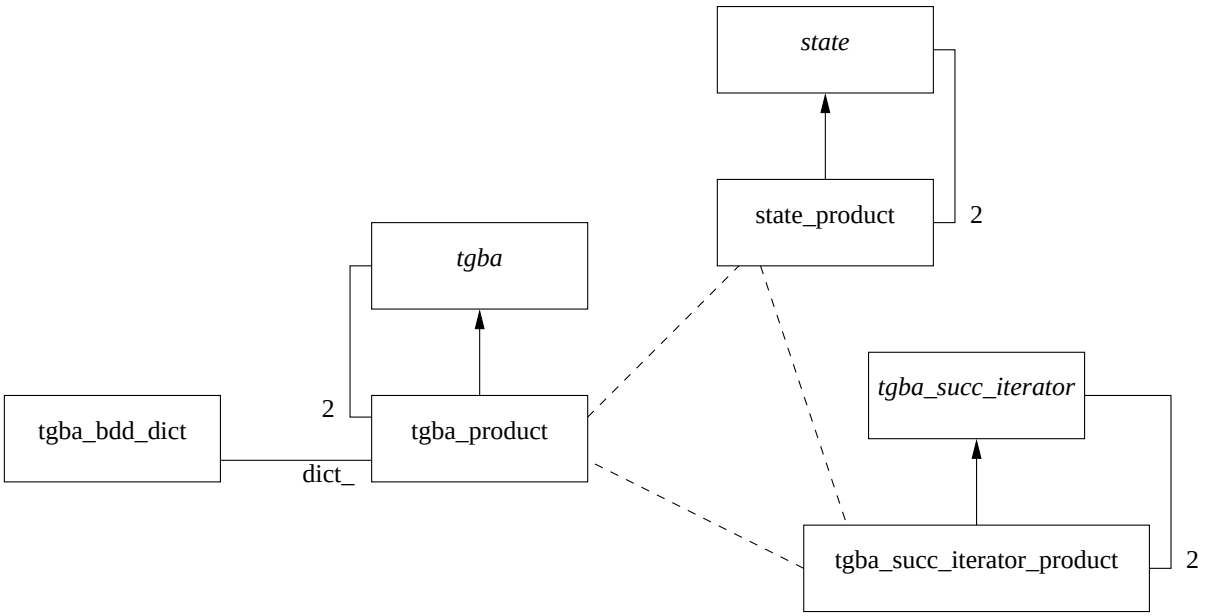
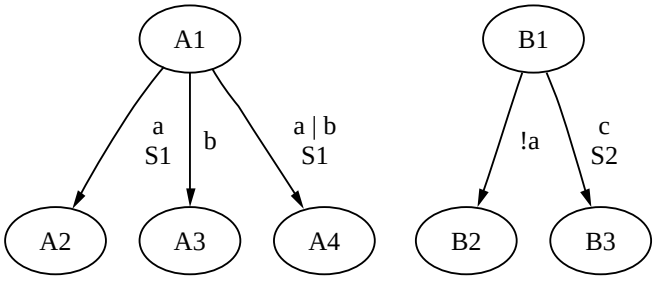
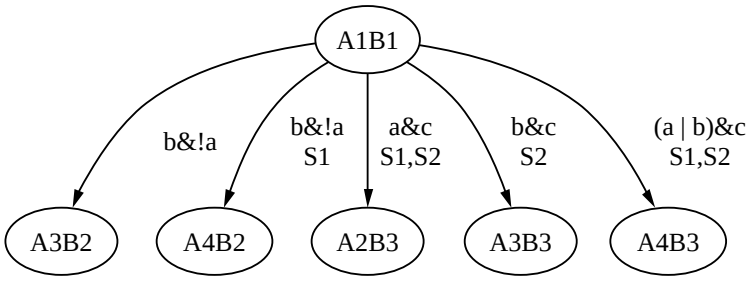


FIG. 4.11 – `tgba_prod` et classes collaboratrices

des `tgba_succ_iterator` construits pour les états correspondants de chaque opérande. Cet itérateur « produit » effectue un produit entre les deux itérateurs « opérandes » de façon à respecter la définition donnée section 3.4 ; grosso modo, il s'agit de faire un produit cartésien des transitions de chaque automate tout en retirant du résultat les transitions qui seraient étiquetées par $\bot$. La figure 4.12 donne un exemple de produit. Dans le produit il n'existe pas d'état A_2B_2 car sa transition devrait être étiquetée par $a \wedge \neg a = \bot$.



(a)



(b)

FIG. 4.12 – (a) deux automates (b) leur produit synchronisé.

Ces produits peuvent être imbriqués comme par exemple dans la figure 4.13. Dans ce cas, le calcul des successeurs d'un état par l'automate « racine » (P_1) déclenche le calcul à la volée des successeurs des automates produits intermédiaires (P_2 et P_3).

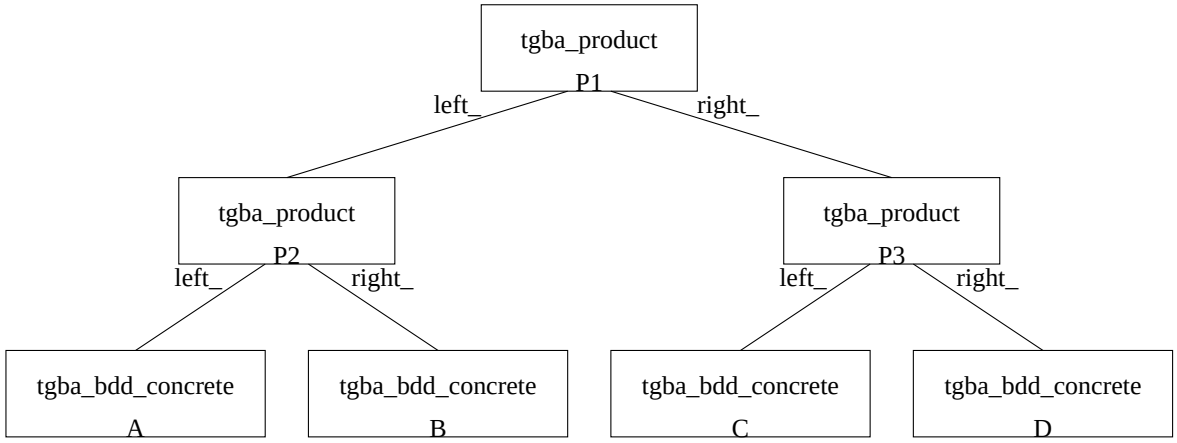


FIG. 4.13 – Produits imbriqués : $(A \otimes B) \otimes (C \otimes D)$.

Un tel calcul imbriqué va peut-être générer beaucoup d'états inutiles. Par exemple supposons que l'on demande à P_1 les successeurs de l'état $s = (s_2, s_3) = ((A_7, B_{23}), (C_{14}, D_{49}))$, sachant que les successeurs des états A_7, B_{23}, C_{14}, et D_{49} dans chacun des automates A, B, C, et D sont ceux représentés figure 4.14.

Le calcul des successeurs de s va provoquer le calcul des successeurs de $s_2 = (A_7, B_{23})$ et $s_3 = (C_{14}, D_{49})$ dans les automates P_2 et P_3. Il y a 9 successeurs pour s_2 et 6 successeurs pour s_3 : à chaque fois il s'agit du produit cartésien des transitions sans simplification possible (les ensembles de variables utilisés sont indépendants). Maintenant, pour

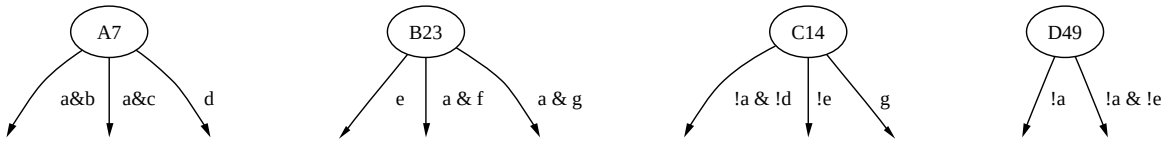


FIG. 4.14 – Besoin d'optimisation lors du calcul des successeurs.

le calcul de s au niveau de P_1, l'itérateur doit réaliser le produit cartésien des deux groupes de 9 et 6 transitions, soit 54 transitions à explorer. Il se trouve que sur ces 54 transitions, une seule est étiquetée par une formule non fausse, elle correspond aux conditions $\neg a \wedge d \wedge e \wedge g$. L'exploration des 53 autres transitions a été faite inutilement. Le cas est d'autant plus critique que lorsque le produit est utilisé pour faire du *model checking* car alors l'un des opérandes — l'automate correspondant au système — affecte toujours une valeur à *chacune* des propositions atomiques.

Une façon de simplifier ces calculs est d'informer chaque itérateur des conditions qui sont valables dans les autres. Cette optimisation est réalisable grâce à la fonction `tgba::support_conditions()` et aux arguments facultatifs de `tgba::succ_iter()` mentionnés section 4.2.3.5.

`a->support_conditions(s)` retourne une formule BDD décrivant des conditions nécessaires (mais pas forcément suffisantes) au franchissement des transitions sortantes de l'état `s` (de l'automate `a`) peut-être franchie. Puisqu'il s'agit de conditions nécessaires le calcul peut être défini sur tout automate : au pire, lorsqu'une telle information est difficile à calculer, il suffit de retourner $\top$[9].

- Dans un `tgba_bdd_concrete`, le calcul de `tgba::compute_support_conditions()` se fait à partir de l'équation 4.9 en supprimant par quantification existentielle toutes les variables de type Now[·] ou Next[·].
- Dans un `tgba_explicit`, le résultat de `tgba_explicit::compute_support_conditions()` s'obtient en calculant explicitement la disjonction des étiquettes des transitions sortantes, ce résultat pourrait[10] être conservé sur chaque nœud pour éviter de le recalculer.
- Dans un `tgba_produit`, `tgba_produit::compute_support_conditions()` effectue simplement la conjonction des conditions de chaque opérande.

Le tableau 4.1 donne des valeurs possibles pour les appels à `support_conditions()` à chaque étape du calcul des successeurs de s.

`a->support_conditions(a7)`	$: (a \wedge (b \vee c)) \vee d$
`b->support_conditions(b23)`	$: e \vee (a \wedge (f \vee g))$
`c->support_conditions(c14)`	$: (\neg a \wedge \neg d) \vee \neg e \vee g$
`d->support_conditions(d49)`	$: \neg a$
`p3->support_conditions(s3)`	$: \neg a \wedge (\neg d \vee \neg e \vee g)$
`p2->support_conditions(s2)`	$: (d \wedge e) \vee (a \wedge (b \vee c \vee d) \wedge (e \vee f \vee g)$
`p1->support_conditions(s)`	$: \neg a \wedge d \wedge e \wedge g$

TAB. 4.1 – Calcul des conditions supportant $s = (s_2, s_3) = ((A_7, B_{23}), (C_{14}, D_{49}))$ dans le produit de la figure 4.13 avec les états de la figure 4.14.

La méthode `support_conditions()` peut-être utilisée par les implémentations de `tgba::succ_iter()` pour restreindre les transitions couvertes par l'itérateur créé à celles qui sont compatibles avec les conditions retournées par `support_conditions()`. Par exemple les successeurs de a_7, sachant que $\neg a \wedge d \wedge e \wedge g$ peuvent ce limiter à l'arc étiqueté par d.

Plutôt que d'appeler `support_conditions()` une fois par état (du produit) puis de passer ce résultat lors de l'appel à `succ_iter()` (récursivement), nous avons décidé de fonctionner dans l'autre sens : c'est la fonction `succ_iter()` qui va appeler `support_conditions()` si elle le souhaite. De cette façon, d'une part l'ajout d'autres méthodes similaires à `support_conditions()` pour simplifier les produits ne se traduira pas par l'ajout d'un nouveau paramètre à `succ_iter()`, et d'autre part les valeurs de chacune de ces fonctions ne sont calculées que si elles sont utilisées.

Les résultats des appels à `support_conditions()` sont mis en cache à chaque étape du produit, donc les appels répétitifs de `support_conditions()` pour le même état, qui sont la contre partie de cette architecture, ne sont pas un problème.

[9] Cela pourrait être l'implémentation par défaut de `tgba::compute_support_conditions()`, mais il semble plus judicieux de ne pas fournir de telle implémentation pour forcer les sous-classes de `tgba` à définir la méthode.

[10] Ce n'est pas fait actuellement.

La définition de `tgba::succ_iter()`, telle qu'elle a été présentée section 4.2.3.5, fait apparaître deux arguments qui sont précisément utilisés dans le cadre d'un produit, pour permettre à `succ_iter()` de réclamer des informations supplémentaires (telles que `support_conditions()`) à l'automate « racine » du produit.

```
virtual tgba_succ_iterator*
succ_iter(const state* local_state,
          const state* global_state = 0,
          const tgba* global_automaton = 0) const = 0;
```

Le paramètre `local_state` est l'état de l'automate pour lequel on souhaite calculer les successeurs. Dans un produit, `global_state` est l'état de l'automate racine du produit `global_automaton` pour lequel on cherche à calculer les successeurs de `local_state`. Il est donc possible d'optimiser ce calcul local en fonction d'informations globales au produit.

Afin d'illustrer l'utilisation de ces arguments, voici comment `tgba_bdd_concrete` implémente cette méthode.

```
1  tgba_succ_iterator_concrete*
2  tgba_bdd_concrete::succ_iter(const state* state,
3                               const state* global_state,
4                               const tgba* global_automaton) const
5  {
6    const state_bdd* s = dynamic_cast<const state_bdd*>(state);
7    assert(s);
8    bdd succ_set = data_.relation & s->as_bdd();
9    // If we are in a product, inject the local conditions of
10   // all other automata to limit the number of successors.
11   if (global_automaton)
12     {
13       bdd varused = bdd_support(succ_set);
14       bdd all_conds = global_automaton->support_conditions(global_state);
15       succ_set = bdd_appexcomp(succ_set, all_conds, bddop_and, varused);
16     }
17   return new tgba_succ_iterator_concrete(data_, succ_set);
18 }
```

La ligne 8 correspond exactement à l'équation 4.9. Les conditions supportant l'état global du produit calculé ligne 14 permettent de restreindre l'ensemble des successeurs ligne 15. Cette restriction se fait en calculant la conjonction $succ\_set \wedge global\_conds$ puis en supprimant les variables qui n'intervenaient pas dans $succ\_set$ par quantification existentielle[11]. La conjonction et la quantification existentielle peuvent être combinées en une seule opération BDD. On retrouve ici la fonction `bdd_appexcomp` ajoutée à BuDDy pour l'occasion (cf. section 4.2.1).

La figure 4.15 montre la création d'un itérateur sur les successeurs de l'état `s`. L'appel `p1->succ_iter(s)`, qui demande la création de cet itérateur provoque la création d'itérateurs sur les états `s2` et `s3` des automates `p2` et `p3` respectivement. Récursivement, des itérateurs sont créés sur les états correspondants des automates A, B, C, et D. Lors de la création des itérateurs pour ces quatre dernier états, la fonction `p1->support_conditions(s)` est appelée : on voit sur le diagramme de séquence que seul le premier appel de `support_conditions` provoque des calculs récursifs, mais que les appels suivants sont cachés.

Une autre opération relative aux automates produits est la projection d'un état (issu d'un automate produit) sur un automate (opérande du produit). Par exemple la projection de s sur l'automate A est l'état a_7. Cette opération est une méthode de la classe `tgba` remplacée dans la classe `tgba_product`.

```
state*
tgba::project_state(const state* s, const tgba* t) const
{
  if (t == this)
    return s->clone();
  return 0;
}

state*
tgba_product::project_state(const state* s, const tgba* t) const
{
```

[11] Sans cette quantification il est facile de produire *plus* de successeurs en appliquant la contrainte `global_conds`. Par exemple si $succ\_set = a \vee b$ et $global\_conds = c \vee d$, la conjonction correspond à 4 transitions et non plus deux.

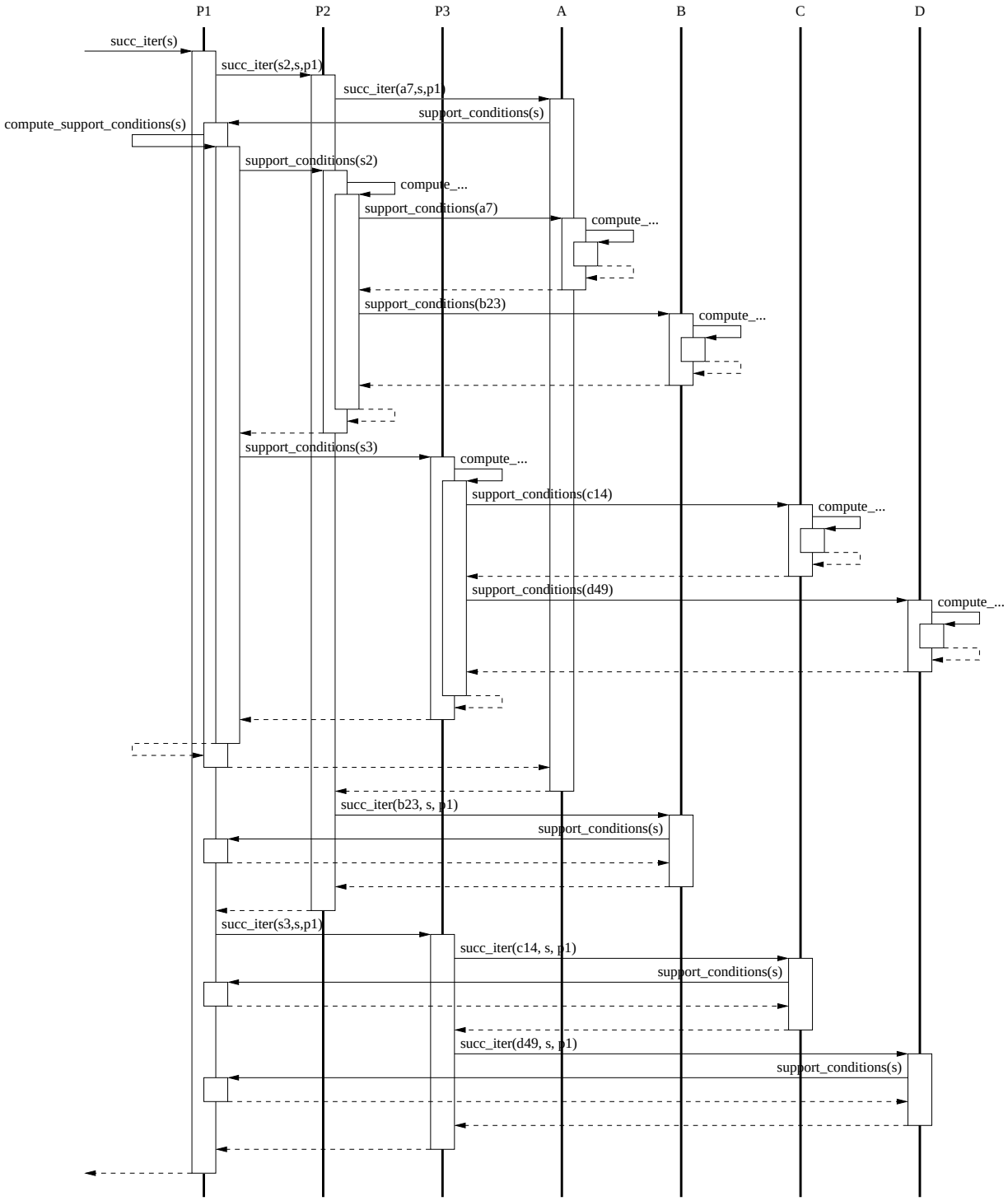


FIG. 4.15 – Diagramme de séquence de l'appel de `p1->succ_iter(s)`.

```
    const state_product* s2 = dynamic_cast<const state_product*>(s);
    assert(s2);
    if (t == this)
      return s2->clone();
    state* res = left_->project_state(s2->left(), t);
    if (res)
      return res;
    return right_->project_state(s2->right(), t);
  }
```

Ainsi, si la cible de la projection n'est pas le `tgba_product` lui-même il essaye de le projeter sur son opérande gauche ou droite, et l'opération se déroule ainsi récursivement.

Enfin, la dernière méthode commune à tous les `tgba`, mais dont la motivation provient de son emploi dans les produits est `neg_accepting_conditions()`.

Lorsqu'un automate A possède deux ensembles d'acceptation $\mathrm{Acc}[f]$ et $\mathrm{Acc}[g]$, ses transitions sont étiquetées par des sous-ensembles de $\{\mathrm{Acc}[f] \wedge \neg\mathrm{Acc}[g], \neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g]\}$. Un autre automate B utilisant l'ensemble d'acceptation $\mathrm{Acc}[h]$ verra ses transitions étiquetées par des sous-ensembles de $\{\mathrm{Acc}[h]\}$ (Ces deux ensembles sont ceux retournés par la méthode `all_accepting_conditions()`.) Le produit de ces deux automates devra étiqueter ses transitions par des sous-ensembles de $\{\mathrm{Acc}[f] \wedge \neg\mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h], \neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h], \neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h]\}$

En pratique, tous les éléments des ensembles apparaissant dans un automate doivent être complétés par la négation des variables $\mathrm{Acc}[\cdot]$ de l'autre automate. Ainsi lorsqu'on synchronise une transition de A étiquetée par $\neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g]$ avec une transition de B étiquetée par $\mathrm{Acc}[h]$, les étapes suivantes sont effectuées
– l'étiquette de la transition de A est complétée par les négations des $\mathrm{Acc}[\cdot]$ de B :

$$(\neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g]) \wedge (\neg\mathrm{Acc}[h])$$

– l'étiquette de la transition de B est complétée par les négations des $\mathrm{Acc}[\cdot]$ de A :

$$(\mathrm{Acc}[h]) \wedge (\neg\mathrm{Acc}[f] \wedge \neg\mathrm{Acc}[g])$$

– enfin ces deux étiquettes sont réunies pour la transition synchronisée :

$$(\neg\mathrm{Acc}[f] \wedge \mathrm{Acc}[g] \wedge \neg\mathrm{Acc}[h]) \vee (\mathrm{Acc}[h] \wedge \neg\mathrm{Acc}[f] \wedge \neg\mathrm{Acc}[g])$$

La méthode `neg_accepting_conditions()` fournit donc les compléments utilisés par les deux premières opérations.

Lorsque les deux automates partagent certaines variables $\mathrm{Acc}[\cdot]$, il faut évidement retirer ces variables du résultat de `neg_accepting_conditions()` avant de compléter les étiquettes.

4.2.3.10 `tgba_tba_proxy`

La classe `tgba_tba_proxy` dégénéralise un automate généralisé à la volée. L'automate construit est toujours un `tgba_bdd_concrete`, mais il utilise exactement un ensemble d'acceptation, quelque soit le nombre d'ensemble utilisés par l'automate d'origine[12].

Outre l'unique ensemble d'acceptation, l'automate `tgba_tba_proxy` possède la particularité suivante : si un état possède une transition sortante appartenant à l'ensemble d'acceptation, alors toutes les transitions sortantes de cet état sont aussi incluses dans l'ensemble d'acceptation. Cela revient à dire que *l'état* est acceptant.

En d'autre termes, un `tgba_tba_proxy` peut être vu comme un automate de Büchi classique dans lequel les propositions portent sur les transitions et les conditions d'acceptation sur les états. Ce sont les types d'automates utilisés par de nombreux *model checkers*, et ils sont donc prérequis par de nombreux algorithmes. La classe `tgba_tba_proxy` offre une méthode qui indique si un état est acceptant, afin d'aider l'écriture de tels algorithmes.

```
bool state_is_accepting(const state* state) const;
```

L'algorithme utilisé est celui présenté par Couvreur [20] : la synchronisation avec un automate compteur. En pratique l'automate compteur n'est pas représenté comme un automate que l'on synchronise explicitement, car le calcul est beaucoup plus simple.

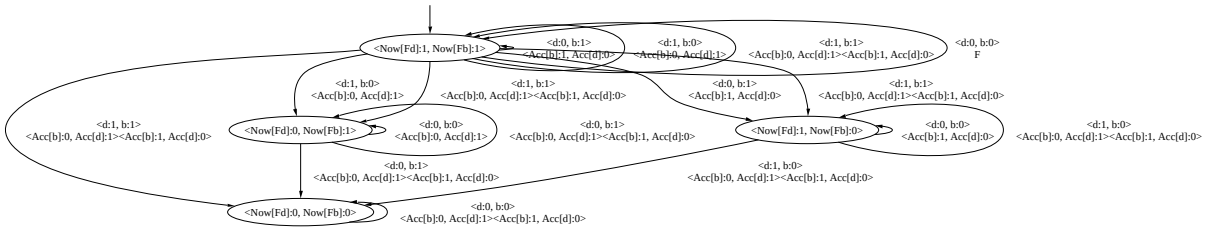
Les M ensembles d'acceptation de l'automate d'origine, plus la nouvelle condition d'acceptation de l'automate résultat, baptisée $\top$ (d'un point de vue ensembliste, $A_\top$ est l'ensemble de toutes les transitions), sont tous ordonnés dans

[12] Il faudrait évidement optimiser cela dans le cas où $M \in \{0, 1\}$. Cela reste à faire.

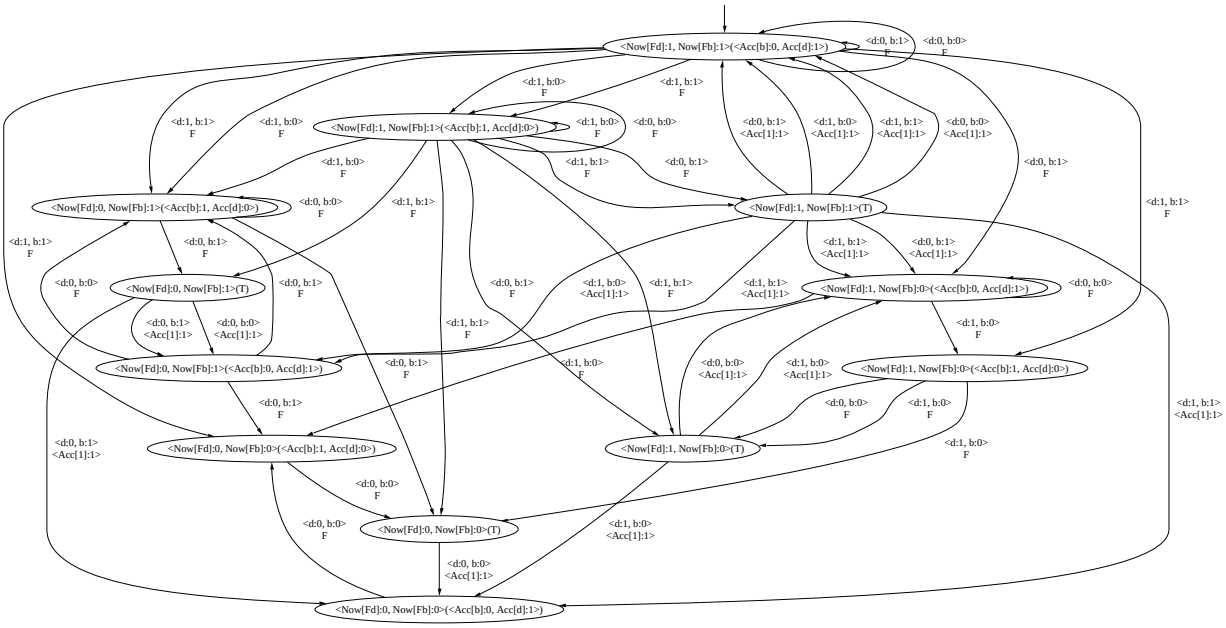
une liste circulaire $A_1, A_2, \ldots, A_M, A_\top, A_1, A_2 \ldots$. Chaque état de l'automate résultat est étiqueté par une paire (s, A), dans laquelle s est l'état correspondant de l'automate d'origine, et A l'un des ensembles d'acceptation de la liste. Si $s \xrightarrow{t} u$ dans l'automate d'origine, alors $(s, A) \xrightarrow{t} (u, B)$, et $B = A$ si $t \notin A$, autrement B est l'ensemble d'acceptation suivant dans la liste circulaire.

Le travail de dégénéralisation est donc effectué par l'itérateur sur les successeurs. Celui-ci suit simplement l'itérateur sur l'état de l'automate d'origine, et change l'ensemble d'acceptation de l'état destination d'une transition lorsque celle-ci appartient à l'ensemble d'acceptation de l'état courant. Les conditions d'acceptation portant sur la transition ne sont non fausses que si l'état est étiqueté par $A_\top$.

Les figures 4.16 et 4.17 montre des exemples de dégénéralisation. Les ensemble d'acceptation associés à chaque état sont notés entre parenthèses. Il est visible que celles les transitions sortant des états étiquetés par $A_\top$ sont acceptantes.



(a)



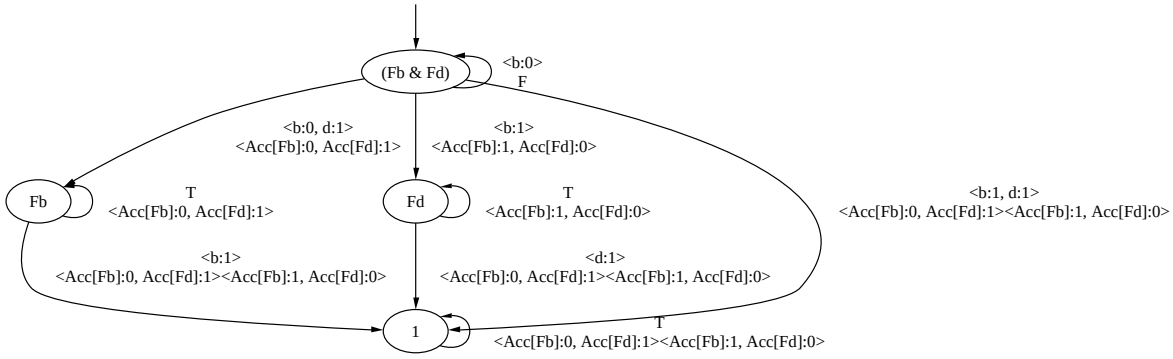
(b)

FIG. 4.16 – Traduction avec `ltl_to_tgba_lacim()` et dégénéralisation avec `tgba_tba_proxy`. (a) La traduction de $\mathsf{F}\, b \wedge \mathsf{F}\, d$. (b) Sa version dégénéralisée.

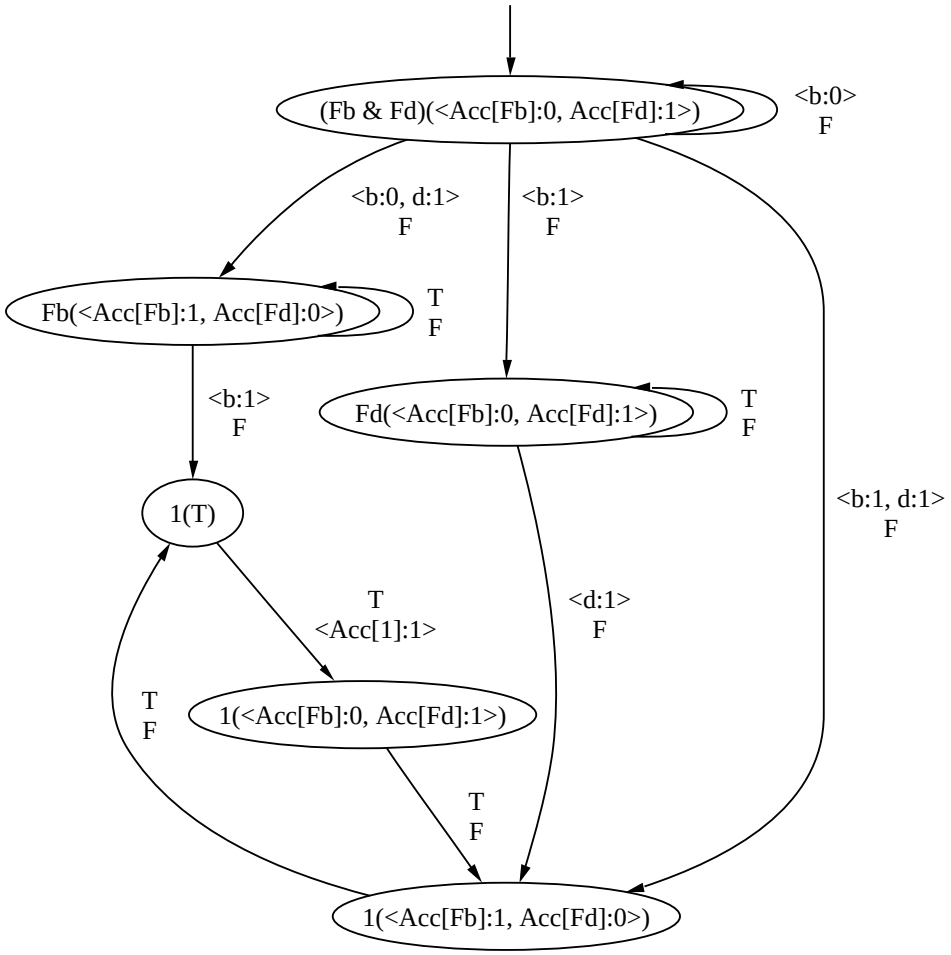
Si l'on note M le nombre d'ensembles d'acceptation de l'automate d'origine, et N son nombre d'états, l'automate dégénéralisé par cette méthode possède au plus $M \times (1 + \max(N, 1))$ états (idem pour les transitions). Ce pire cas est atteint dans l'exemple figure 4.16, où 4 états sont devenus 12.

4.2.4 Algorithmes

Cette section décrit les algorithmes de `libspot` manipulant des `tgba`. Certains de ces algorithmes proviennent d'articles récents, que nous déjà évoqués chapitre 2, mais comme nos implémentations différent parfois des implémentations publiées nous détaillons les algorithmes en plus de préciser leur interface dans la bibliothèque. D'autre part, les algorithmes ainsi présentés laissent transparaître l'utilisation des objets de la bibliothèque.



(a)



(b)

FIG. 4.17 – Traduction avec `ltl_to_tgba_fm()` et dégénéralisation avec `tgba_tba_proxy`. (a) La traduction de $\mathsf{F}\, b \wedge \mathsf{F}\, d$. (b) Sa version dégénéralisée.

4.2.4.1 ltl_to_tgba_lacim

```
tgba_bdd_concrete* ltl_to_tgba_lacim(const ltl::formula* f,
                                     bdd_dict* dict);
```

`ltl_to_tgba_lacim()` implémente l'algorithme de Couvreur/LaCIM [21] (introduit en section 2.3.1.9) pour transformer une `formula` en un `tgba_bdd_concrete`. La conversion se fait avec un `ltl::const_visitor` en traversant récursivement la formule, et retourne un triplet de BDD $\langle q, c, A\rangle$ dans lequel q désigne l'état initial (à l'aide de variables Now[·], Next[·], et des propositions atomiques), c désigne des contraintes à observer lors du passage d'un état à ces successeurs (cette fonction lie des variables Now[·], Next[·], et des propositions atomiques), et A est un ensemble de paires (a, w) où a est un ensemble d'acceptation et w les conditions (en termes de variables Now[·] et des propositions atomiques) sous lesquelles une transition appartient à a.

La réécriture, dérivée du tableau 2.5 page 30 se déroule comme suit. p désigne une proposition atomique, f et g des formules LTL quelconques.

$$
\begin{aligned}
ltl\_to\_tgba(\top) &= \langle \top, \top, \emptyset\rangle \\
ltl\_to\_tgba(\bot) &= \langle \bot, \top, \emptyset\rangle \\
ltl\_to\_tgba(p) &= \langle p, \top, \emptyset\rangle \\
ltl\_to\_tgba(\neg p) &= \langle \neg p, \top, \emptyset\rangle \\
ltl\_to\_tgba(\bigwedge_{f\in F} f) &= \langle \bigwedge_{f\in F} q_f, \bigwedge_{f\in F} c_f, \bigcup_{f\in F} A_f\rangle \quad \text{où} \quad \forall f \in F,\ \langle q_f, c_f, A_f\rangle = ltl\_to\_tgba(f) \\
ltl\_to\_tgba(\bigvee_{f\in F} f) &= \langle \bigvee_{f\in F} q_f, \bigwedge_{f\in F} c_f, \bigcup_{f\in F} A_f\rangle \quad \text{où} \quad \forall f \in F,\ \langle q_f, c_f, A_f\rangle = ltl\_to\_tgba(f) \\
ltl\_to\_tgba(f \,\mathsf{U}\, g) &= \langle \text{Now}[f \,\mathsf{U}\, g], c_f \wedge c_g \wedge \text{Now}[f \,\mathsf{U}\, g] \Leftrightarrow (q_g \vee (q_f \wedge \text{Next}[f \,\mathsf{U}\, g])), \\
&\qquad A_f \cup A_g \cup \{(\text{Acc}[g], q_g \wedge \neg\text{Now}[f \,\mathsf{U}\, g])\}\rangle \\
&\text{où} \quad \langle q_f, c_f, A_f\rangle = ltl\_to\_tgba(f) \quad \text{et} \quad \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g) \\
ltl\_to\_tgba(\mathsf{F}\, g) &= \langle \text{Now}[\mathsf{F}\, g], c_g \wedge \text{Now}[\mathsf{F}\, g] \Leftrightarrow (q_g \vee \text{Next}[f \,\mathsf{U}\, g]), A_g \cup \{(\text{Acc}[g], q_g \wedge \neg\text{Now}[Fg])\}\rangle \\
&\text{où} \quad \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g) \\
ltl\_to\_tgba(f \,\mathsf{R}\, g) &= \langle \text{Now}[f \,\mathsf{R}\, g], c_f \wedge c_g \wedge \text{Now}[f \,\mathsf{R}\, g] \Leftrightarrow (q_g \wedge (q_f \vee \text{Next}[f \,\mathsf{R}\, g])), A_f \cup A_g\rangle \\
&\text{où} \quad \langle q_f, c_f, A_f\rangle = ltl\_to\_tgba(f) \quad \text{et} \quad \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g) \\
ltl\_to\_tgba(\mathsf{G}\, g) &= \langle \text{Now}[\mathsf{G}\, g], c_g \wedge \text{Now}[\mathsf{G}\, g] \Leftrightarrow (q_g \wedge \text{Next}[\mathsf{G}\, g]), A_g\rangle \\
&\text{où} \quad \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g)
\end{aligned}
\tag{4.12}
$$

Les réécritures pour les opérateurs U, F, R, et G sont basées sur les équations 2.1 page 10.

Une fois qu'une formule f, a été récrite en $\langle q_f, c_f, A_f\rangle$, l'automate `tgba_bdd_concrete` se définit en prenant q_f comme état initial, c_f comme relation de transition, et la relation d'acception construite de façon à respecter toutes les paires portant sur les mêmes conditions d'acceptation :

$$
\bigvee_{l\in\{a|(a,\cdot)\in A_f\}} \left(l \wedge \bigwedge_{\substack{g\in\{a|(a,\cdot)\in A_f\}\\ g\neq l}} g \wedge \bigwedge_{h\in\{w|(l,w)\in A_f\}} h \right)
$$

Par exemple $A = \{(a, w_1), (b, w_2), (a, w_3)\}$ devient $(a\wedge\neg b\wedge w_1\wedge w_3)\vee(\neg a\wedge b\wedge w_2)$. C'est-à-dire qu'une transition appartient à l'ensemble d'acceptation a (désigné par $a \wedge \neg b$) si et seulement si les conditions w_1 et w_3 sont vérifiées. De même une transition appartient à l'ensemble d'acceptation b si et seulement si la condition w_2 est vérifiée.

Cette traduction repose sur le fonctionnement des automates `tgba_bdd_concrete` expliqué section 4.2.3.8. En particulier la façon de calculer les successeurs qui satisfont la relation de transition, équation 4.9.

Couvreur [21] propose une optimisation qui s'applique lorsque la formule à traduire est de type $\mathsf{G}\, f$. Cette optimisation ne peut être effectuée récursivement, nous notons $ltl\_to\_tgba_1$ pour montrer qu'elle ne s'applique qu'à la racine de l'arbre de syntaxe.

$$
\begin{aligned}
ltl\_to\_tgba_1(\mathsf{G}\, g) &= \langle q_g, c_g \wedge q_g, A_g\rangle \\
&\text{où} \quad \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g)
\end{aligned}
$$

L'idée est simple : comme $\mathsf{G}\,g$ n'est pas une sous-formule, mais la formule complète, g doit être vérifiée à tout moment par l'automate. Il suffit donc d'ajouter q_g comme contrainte à la relation de transition. Ceci supprime l'introduction des variables Now$[g]$ et Next$[g]$, ce qui est toujours souhaitable car en dehors de l'état initial tout état est une affectation de l'ensemble des variables de type Now$[\cdot]$: il y a au plus $1 + 2^{|\text{Now}[\cdot]|}$ états.

Dans `libspot`, nous avons généralisé cette optimisation en

$$\begin{aligned} ltl\_to\_tgba_1(f \wedge \bigwedge_{g\in G} \mathsf{G}\,g) = \langle q_f \wedge \bigwedge_{g\in G} q_g, c_f \wedge \bigwedge_{g\in G} q_g, A_f \cup \bigcup_{g\in G} A_g\rangle \\ \text{où} \quad \langle q_f, c_f, A_f\rangle = ltl\_to\_tgba(f) \\ \text{et} \quad \forall g \in G,\ \langle q_g, c_g, A_g\rangle = ltl\_to\_tgba(g) \end{aligned}$$

La figure 4.7(a) page 66 montre la traduction de $\mathsf{X}\,a \wedge (b\,\mathsf{U}\,\neg a)$ avec cette méthode.

4.2.4.2 `ltl_to_tgba_fm`

```
tgba_explicit* ltl_to_tgba_fm(const ltl::formula* f, bdd_dict* dict);
```

La fonction `ltl_to_tgba_fm()` réalise la traduction d'une `formula` en `tgba_explicit` selon l'algorithme de Couvreur/FM [20] introduit section 2.3.1.5.

Cette traduction fait intervenir la réécriture suivante (implémentée sous la forme d'un `ltl::const_visitor`), pour transformer une `formula` en un BDD.

$$\begin{aligned} rewrite(f\,\mathsf{U}\,g) &= rewrite(g) \vee (\text{Acc}[g] \wedge rewrite(f) \wedge \text{Next}[f\,\mathsf{U}\,g]) \\ rewrite(\mathsf{F}\,g) &= rewrite(g) \vee (\text{Acc}[g] \wedge \text{Next}[\mathsf{F}\,g]) \\ rewrite(f\,\mathsf{R}\,g) &= rewrite(g) \wedge (rewrite(f) \vee \text{Next}[f\,\mathsf{R}\,g]) \\ rewrite(\mathsf{G}\,g) &= rewrite(g) \wedge \text{Next}[\mathsf{G}\,g] \\ rewrite(\bigwedge_{f\in F} f) &= \bigwedge_{f\in F} rewrite(f) \\ rewrite(\bigvee_{f\in F} f) &= \bigvee_{f\in F} rewrite(f) \\ rewrite(f \vee g) &= rewrite(f) \vee rewrite(g) \\ rewrite(p) &= \text{Var}[p] \\ rewrite(\neg p) &= \neg\text{Var}[p] \\ rewrite(\top) &= \top \\ rewrite(\bot) &= \bot \end{aligned} \tag{4.13}$$

Ici aussi, les réécritures pour les opérateurs U, F, R, et G sont basées sur les équations 2.1 page 10.

Trois types de variables apparaissent dans le BDD produit : les variables Var$[\cdot]$ qui encodent les propositions atomiques, les variables Next$[\cdot]$ traduisent des formules (non récrites) qui devront être vérifiées à l'instant suivant, et des variables Acc$[\cdot]$ qui encodent une promesse (Acc$[g]$ est la promesse de vérifier g dans le futur). À la différence de Couvreur [20], nous associons à $f\,\mathsf{U}\,g$ la variable Acc$[g]$ et non Acc$[f\,\mathsf{U}\,g]$, cela diminue le nombre d'ensembles d'acceptation.

L'algorithme de traduction est présenté figure 4.18. La figure 4.19 montre l'automate obtenu lors de la traduction de $\mathsf{X}\,a \wedge (b\,\mathsf{U}\,\neg a)$. Il faut le comparer avec la figure 2.13(b) page 26. On peut noter que certains états sont reliés par des paires de transitions étiquetées l'une par $\neg a \wedge \neg b$ et l'autre par $\neg a \wedge b$, alors qu'une transition étiquetée par $\neg a$ aurait suffit (c'est d'ailleurs ainsi que la figure 2.13(b) montre l'automate). Il s'agit d'un cas où BuDDy et Spot n'ont pas réussi à simplifier les impliquants de l'équation 4.9 page 68.

4.2.4.3 `emptiness_check`

Cette section concerne l'implémentation de l'algorithme informellement présenté en section 2.4. Cette implémentation repose sur le calcul de composante fortement connexe d'un `tgba`. Ces dernières sont représentées par la structure `connected_component`.

```
struct connected_component
{
  int index;
```

```
ltl_to_tgba_fm(f)
{
  f' ← negative_normal_form(unabbreviate_logic(f));
  todo ← {f'};
  seen ← {f'};
  a ← new tgba_explicit;
  while (todo ≠ ∅)
    {
      g ← todo.remove_one();
      b ← rewrite(g);
      forall i ∈ prime_implicants_of(b)
        {
          Put i in the form ⋀_{v∈V} Var[v] ∧ ⋀_{a∈A} Acc[a] ∧ ⋀_{n∈N} Next[n];
          dest ← ⋀_{n∈N} n;
          t ← a.create_transition(g, dest);
          t.add_condition(⋀_{v∈V} Var[v]);
          t.add_accepting_condition(⋀_{a∈A} Acc[a]);
          if (dest ∉ seen)
            {
              todo.insert(dest);
              seen.insert(dest);
            }
        }
    }
  // Turn all promises into real accepting conditions.
  a.complement_all_accepting_conditions();
  return a;
}
```

FIG. 4.18 – Algorithme `ltl_to_tgba_fm`.

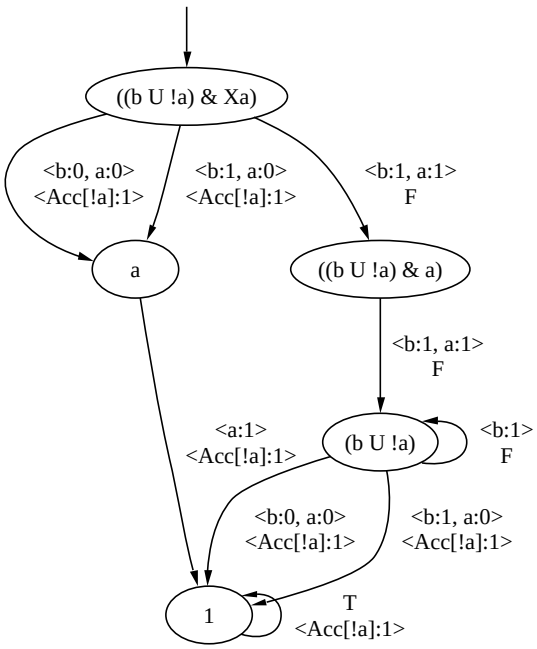


FIG. 4.19 – `ltl_to_tgba_fm`$(\mathsf{X}\, a \wedge (b \,\mathsf{U}\, \neg a))$.

```
    bdd condition;
    set_of_states state_set;
    //...
  };
```

L'attribut `index` joue le rôle de l'identifiant pour une composante fortement connexe (CFC) dans un `tgba`. Lors de l'*emptiness check*, l'`index` fait référence au premier état de cette CFC rencontré lors du parcours en profondeur. Cet état est alors appelé *racine de composante fortement connexe*. Ainsi, l'`index` permet d'énumérer les composantes fortement connexes maximales d'un graphe construites lors d'une recherche en profondeur.

`condition` est un BDD représentant l'union de toutes les conditions d'acceptation des arcs considérés au sein d'une CFC. Dans l'algorithme cet attribut est au centre du test de *détection de cycle acceptant*.

`state_set` est l'ensemble d'états d'une CFC. Dans l'algorithme du contre-exemple, on justifiera l'importance de cet ensemble.

La classe `emptiness_check` fournit la réponse du test du vide d'un langage accepté par un `tgba` donné. Dans le cas où la réponse est négative, un contre-exemple peut être construit. L'interface d'un `emptiness_check` est la suivante :

```
  class emptiness_check
  {
  public:
    bool tgba_emptiness_check(const spot::tgba* aut_check);
    void counter_example(const spot::tgba* aut_counter);
    void remove_component(const tgba& aut, seen& state_map,
                          const spot::state* start_delete);
    void accepting_path (const spot::tgba* aut_counter,
                         const connected_component& comp_path,
                         const spot::state* start_path, bdd to_accept);
    void complete_cycle(const spot::tgba* aut_counter,
                        const connected_component& comp_path,
                        const state* from_state,const state* to_state);

    std::stack<connected_component> root_component;
    state_sequence prefix;
    cycle_path period;
  private:
    seen H;
    std::stack <bdd> arc_accepting;
    std::stack <pair_state_iter> todo;
    std::vector<state_sequence> vec_sequence;
    //...
  };
```

`root_component` est une pile de CFC construite par la méthode `tgba_emptiness_check()` lors de l'exploration de l'automate. Lorsque `tgba_emptiness_check()` retourne **`true`** (indiquant ainsi que le langage de l'automate testé est non-vide), le sommet de la pile contient une CFC accessible possédant toutes les conditions d'acceptation de l'automate.

Lors du calcul d'un contre-exemple, celui-ci est découpé en deux parties : un préfixe, suivi d'un circuit. Le préfixe est une séquence d'états (attribut `prefix`) ; comme il n'a pas besoin de satisfaire les conditions d'acceptation de l'automate, n'importe quelle séquence de transitions passant par ces états peut-être utilisée pour former le début d'un contre exemple complet. Le circuit (attribut `period`) doit quant à lui vérifier les conditions d'acceptation de l'automate, il est donc représenté par une séquence de paires (états, formule propositionnelle).

Les méthodes de cette interface sont les implémentations des algorithmes centraux de l'*emptiness check* et du *contre-exemple*.

La méthode `tgba_emptiness_check()` détermine si le langage accepté par un automate de Büchi généralisé étiqueté sur les transitions considéré est vide. Elle retourne **`true`** dans ce cas. Autrement, elle laisse une pile de CFC (`root_component`) à partir de laquelle `counter_example()` peut extraire un contre-exemple.

`tgba_emptiness_check()` implémente un algorithme développé par Couvreur [20]. Celui-ci s'interrompt dès qu'il trouve une composante fortement connexe (pas nécessairement maximale) accessible depuis l'état initial et vérifiant les conditions d'acceptation de Büchi généralisées.

L'algorithme est compatible avec une construction à la volée de l'automate à tester (c'est-à-dire qu'il peut se contenter de l'interface d'un TGBA). Et seules les CFC « *utiles* » — c'est-à-dire celles qui peuvent conduire à une CFC possédant toutes les conditions d'acceptation — rencontrées lors du parcours en profondeur sont conservées en mémoire.

Les structures de données de notre algorithme suffisent pour conserver les informations concernant la visite d'un état lors du parcours en profondeur et pour construire la pile de composantes fortement connexes rencontrées.

- $nbstate$ permet de déterminer l'ordre de la première visite d'un état lors du parcours en profondeur.
- $root\_component$ est une pile de CFC dont les états ne sont pas explicités $\langle r, acc_{\mathcal{C}_r} \rangle$. Elles sont construites avec un ensemble d'état vide. Pour les besoins de cet algorithme, l'index et l'ensemble d'acceptation associé suffisent. Dans l'exécution de notre algorithme, on dépilera $root\_component$ lors d'une détection de cycle dans notre parcours pour ensuite fusionner ces composantes extraites. A terme, nous obtiendrons alors une pile de CFC.
- $arc\_accepting$ est une pile d'ensembles de conditions d'acceptation qui détermine les conditions d'acceptation portées sur les arcs reliant deux CFC construites consécutivement lors du parcours en profondeur.
- H est une table de paires (état, entier) qui enregistre l'ordre de visite des états lors du parcours en profondeur. Nous noterons $H[s]$ l'ordre de visite d'un état s. H ne contient que des états visités, mais certains de ceux-ci peuvent avoir été abandonnés lorsque l'algorithme détecte que leur contribution au langage de l'automate est nulle, dans ce cas $H[s] = -1$. Dans l'algorithme de Tarjan, l'ordre de visite associé à chaque état permet de construire les CFC.
- $stack$ est la pile du parcours en profondeur. Elle contient des couples (état, successeurs non-explorés). En pratique ces successeurs non-explorés sont représentés par un itérateur.

Une description de l'algorithme est donnée figure 4.20.

- Les *lignes 3–7* constituent l'initialisation. On commence par insérer l'état initial de l'automate considéré. La racine de la première CFC est évidement l'état initial et à ce stade du parcours, cette première composante ne contient aucune condition d'acceptation (aucune transition empruntée). Le premier ensemble de conditions d'acceptation dans $arc\_accepting$ est associé à une transition « artificielle » arrivant sur l'état initial.
- *Ligne 8* : si tous les états ont été visités ($stack = \emptyset$) alors le langage reconnu par l'automate est vide.
- *Ligne 11* : deux cas se présentent : Si $succ = \emptyset$, tous les états atteignables depuis q ont été visités. Nous considérons tout d'abord le cas où q est racine d'une CFC (*ligne 16*). A ce stade de l'exécution, l'algorithme n'a pas retourné de résultat et $stack$ n'est pas vide, nous en déduisons que aucune CFC *acceptante* est atteignable depuis q. C'est pourquoi, on peut supprimer tous les états atteignables depuis q. Dans le cas contraire, q est un élément d'une CFC non-maximale qui peut être fusionnée avec d'autre CFC lors du parcours en profondeur.
  Sinon, on distingue deux cas selon la position courante de l'itérateur sur la liste des successeurs possibles. Si le successeur de q n'a pas été visité, nous l'insérons dans la table pour préserver son ordre dans le parcours puis il est considéré comme la racine d'une CFC. A ce stade de l'exécution, cette CFC fait uniquement référence à cet état. Pour assurer le parcours en profondeur et la visite de ses successeurs, nous empilons $stack$. Dans le cas où le successeur a déjà été visité (*ligne 37*), le parcours en profondeur introduit un nouveau cycle de CFC. Nous procédons ensuite à la construction de la CFC représentant ce cycle en fusionnant toutes les CFC y figurant. Cette *fusion* de CFC est réalisée en prenant la réunion des ensembles de conditions d'acceptation des CFC et des arcs ($arc\_accepting$) qui les relient dans le parcours. Nous empilons ainsi les CFC rencontrées. Cette fusion de CFC est représentée figure 4.21.
- *Ligne 49* : lorsqu'une CFC est construite suite à la détection d'un cycle, nous vérifions si elle contient toutes les conditions d'acceptation. Si c'est le cas, le langage de l'automate n'est pas vide.

L'algorithme préserve un certain nombre de propriétés durant son exécution.

propriété 1. *$root\_component$ est composée uniquement de CFC dont la racine (`index`) est l'ordre de visite d'un état dans le parcours en profondeur. L'ordre des racines de CFC est donc celui défini lors du parcours.*

Dans $root\_component$, une CFC C_i est définie par sa racine $root_i$ et son ensemble de conditions d'acceptation acc_i. Dans cette pile nous ne donnons pas une définition explicite d'une CFC car nous ne retenons pas son ensemble d'états. Cependant dans l'algorithme du *contre exemple* cet ensemble devra être donné. La propriété suivante nous permet de retrouver l'ensemble d'états formant les CFC à partir de la table H.

propriété 2. *Si $root\_component = (root_1, acc_1) \cdot (root_2, acc_2) \cdots (root_n, acc_n)$ alors $root_1 = 1$ et on définit explicitement la CFC C_i du graphe courant (sous-graphe défini par l'ensemble des états dans la table dont la valeur est différente de -1) comme suit : $\forall i < n$, $C_i = \{s \in \mathcal{Q} \mid (H[s] \neq -1) \wedge (root_i \leq H_s \leq root_i - 1)\}$ et $C_n = \{s \in \mathcal{Q} \mid (H[s] \neq -1) \wedge (root_n \leq H_s \leq nbstate)\}$.*

propriété 3. *Par construction, si $arc\_accepting = B_1 \cdot B_2 \cdot \ldots \cdot B_n$ alors les CFC de `root_component` sont reliées successivement entre elles dans le graphe courant par des transitions de la forme : $(s_{C_{i-1}}, B_i, order^{-1}(root_i))$ avec $i \in]1, n]$ et $s_{C_{i-1}} \in C_{i-1}$ (autrement dit $order(s_{C_{i-1}}) \in [root_{i-1}, root_i[$. Et comme nous l'avions signalé précédemment, $B_1 = \emptyset$ car il correspond à une transition artificielle arrivant sur l'état initial.*

propriété 4. *L'ensemble des états $\{s \in \mathcal{Q} \mid H[s] = -1\}$ est une union de CFC ne contenant pas toutes les conditions d'acceptation de l'automate.*

```
1  tgba_emptiness_check(⟨AP, Q, δ, q0, F⟩)
2  {
3      nbstate ← 1;
4      stack.push(⟨q0, ⊥, δ(q0)⟩);
5      H[q0] ← 1;
6      root_component.push(⟨q0, ∅⟩);
7      arc_accepting.push(∅);
8      while (stack ≠ ∅)
9        {
10         ⟨q, succ⟩ ← stack.top();
11         if (succ = ∅)
12           {
13             stack.pop();
14             ⟨r, acc_{C_r}⟩ ← root_component.top();
15             if (r = H[q])
16               {
17                 remove_component(q);
18                 root_component.pop();
19                 arc_accepting.pop();
20               }
21           }
22           else
23             {
24               ⟨acc_{qq'}, q'⟩ ← succ.remove_one();
25               if (q' ∉ H)
26                 {
27                   H[q'] ← nbstate;
28                   root_component.push(⟨q', ∅⟩);
29                   arc_accepting.push(acc_{qq'});
30                   stack.push(⟨q', δ(q')⟩);
31                   nbstate ← nbstate + 1;
32                 }
33               else
34                 {
35                   H_{q'} ← H[q'];
36                   if (H_{q'} ≠ −1)
37                     {
38                       ⟨r, acc_{C_r}⟩ ← root_component.top();
39                       root_component.pop();
40                       new_acc ← acc_{qq'} ∪ acc_{C_r};
41                       while (r > H_{q'})
42                         {
43                           ⟨r', acc'_{C_{r'}}⟩ ← root_component.top();
44                           new_acc ← new_acc ∪ acc'_{C_{r'}} ∪ arc_accepting.top();
45                           arc_accepting.pop();
46                           r ← r';
47                         }
48                     }
49                   root_component.push(H_{q'}, new_acc);
50                   if (⋃_{i<|F|}{F_i | F_i ∈ F} = new_acc)
51                       return ⊥;
52                 }
53             }
54       }
55       return ⊤;
56   }
```

FIG. 4.20 – Algorithme *on-the-fly emptiness check* de Couvreur [20].

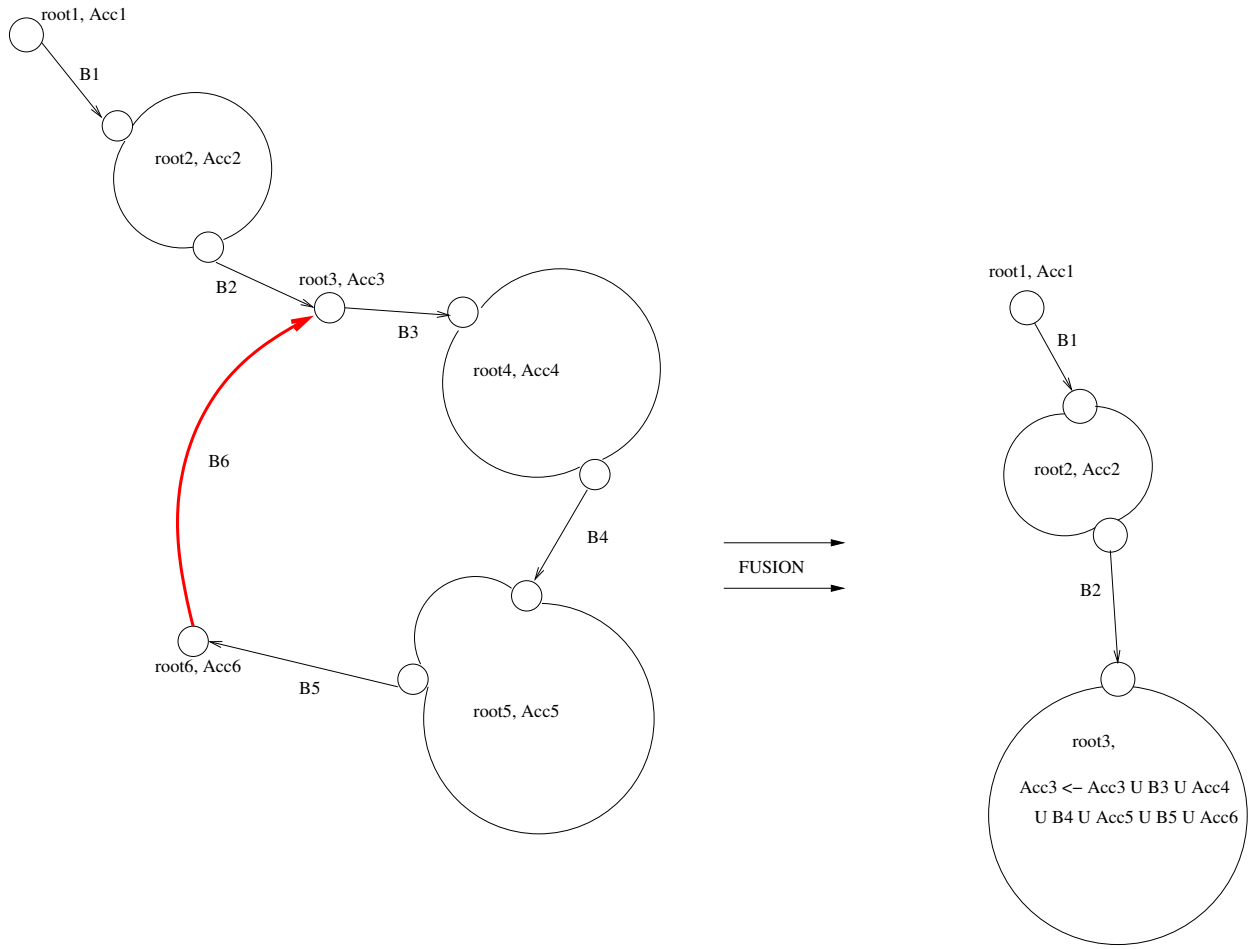


FIG. 4.21 – Évolution de la pile de CFC et de la pile d'ensembles de conditions d'acceptation lors d'une fusion de composantes. La i^{e} CFC empilée est notée $root_i, Acc_i$ et le j^{e} ensemble de conditions d'acceptation empilé est noté B_j. Les flèches symbolisent le parcours en profondeur.

Théorème 2. *Soit A un* `tgba`*. S'il existe au moins une CFC contenant toutes les conditions d'acceptation dans A alors* `tgba_emptiness_check` *retourne* **`false`***. De plus, l'algorithme retourne ce résultat dès la détection de cette CFC sur le graphe courant.*

Idée de la preuve : La démonstration est directe, car elle repose uniquement sur le fait que les propriétés précitées sont bien vérifiées avant ou après les exécutions suivantes :
– Avant d'empiler et de dépiler $stack$ et avant une itération sur la liste des successeurs (ligne 24).
– Après avoir visité un état qui a une valeur différente de -1 dans la table et après avoir itéré sur tout les successeurs d'un état
– Après avoir vérifié si une composante contient toutes les conditions et après avoir appelé `remove`.
Nous illustrons sur l'automate de la figure 4.22 tous les aspects d'une exécution de l'algorithme.

1. L'algorithme considère tout d'abord le chemin $A \longrightarrow B \longrightarrow C \longrightarrow D$. Tous ces états apparaissent comme des nouveaux états lors du parcours en profondeur. Les lignes 3–7 sont exécutées une fois puis l'exécution des lignes 26–32 est répétée pour chaque état rencontré dans ce chemin.
2. Ensuite l'algorithme considère la transition $D \longrightarrow A$. Les lignes 36–51 sont exécutées. $root\_component$ et $arc\_accepting$ sont dépilés pour obtenir la composante $\{A, B, C, D\}$. L'algorithme poursuit car cette composante n'est pas acceptante.
3. L'algorithme considère ensuite le chemin $C \longrightarrow E \longrightarrow F$ en exécutant les lignes 26–33. Puis la transition $F \longrightarrow E$ est empruntée. Nous obtenons alors la composante $\{E, F\}$. Les successeurs de F ont été explorés, les lignes 10–22 sont exécutées. F n'est pas une racine de CFC (il est juste exploré). De la même façon les successeurs de E ont été explorés et E est la racine de $\{E, F\}$. Donc la composante $\{E, F\}$ est supprimée du graphe courant.
4. L'algorithme considère le chemin $C \longrightarrow G \longrightarrow H \longrightarrow G$. Il construit et empile la composante $\{G, H\}$. Elle n'est pas acceptante car son ensemble d'acceptation est $\{Acc[a]\}$. Le chemin $H \longrightarrow I \longrightarrow G$ est considéré et les lignes 35–52 sont exécutées. La composante $\{G, H, I\}$ est construite. Elle n'est pas acceptante car son ensemble d'acceptation est $\{Acc[a], Acc[b]\}$. Le chemin $I \longrightarrow J \longrightarrow G$ est considéré et les lignes 35–52 sont exécutées. La composante $\{G, H, I, J\}$ est construite. Elle est acceptante car son ensemble d'acceptation est $\{Acc[a], Acc[b], Acc[c]\}$. L'algorithme retourne **`false`**.

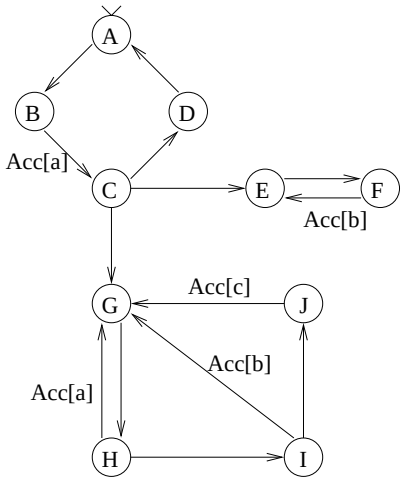


FIG. 4.22 – Un `tgba` avec les conditions d'acceptation $Acc[a], Acc[b], Acc[c]$.

La méthode `counter_example()` peut être appelée lorsque `tgba_emptiness_check()` a retourné **`false`**. Dans ce cas, nous avons une pile de CFC qui vérifient les propriétés ci-dessus. Cette méthode calcule le préfixe d'un contre-exemple. Ce préfixe est une séquence d'états, construite à partir de l'état de la pile de recherche en profondeur ($root\_component$) au moment de l'arrêt de `tgba_emptiness_check()`.

`counter_example()` calcule un chemin passant par chacune des CFC empilées[13]. Pour ce faire, nous avons besoin d'une représentation explicite des composantes connexes. Ces ensembles (que `tgba_emptiness_check()` a laissés vides) peuvent être calculés par un parcours de H.

Soit un état visité s tel que $H[s] \neq -1$. Notons $root\_component = (root_1, acc_1) \cdot (root_2, acc_2) \cdots (root_n, acc_n)$ la pile des CFC. Si $\exists i \in [1, n[$ tel que $root_i \leq H[s] < root_{i+1}$ alors s est un état de la CFC représenté par $(root_i, acc_i)$

[13]Cette interface doit être retravaillé. Pour le moment, l'utilisateur doit appeler `counter_example()` pour calculer le préfixe, et `accepting_path()` pour calculer le circuit acceptant. Ces différentes étapes doivent absolument être dissimulées à l'utilisateur. Il faut que ce dernier puisse réclamer un contre-exemple sans présumer de la façon dont ce calcul est fait, c'est-à-dire qu'on s'attend à ce que `counter_example()` *retourne* directement un example complet.

dans la pile. Si $H[s] > root_n$ alors s est un état de la dernière composante connexe. Nous noterons $\mathcal{C}_i$ l'ensemble des états appartenant à la CFC désignée par $(root_i, acc_i)$ dans $root\_component$.

À partir d'un état d'une CFC $\mathcal{C}_i$, l'algorithme effectue une recherche en largeur pour trouver le chemin le plus court vers la CFC $\mathcal{C}_{i+1}$. Ce procédure est itérée pour trouver un chemin entre l'état initial et $\mathcal{C}_n$.

Les structures de données de l'algorithme 4.23 sont présentées ci-dessous :

– $queue$ est une file FIFO dont les éléments sont des paires d'états et d'itérateurs sur la liste des successeurs ($\langle etat, succ \rangle$).
– H est la table construite par l'exécution précédente de `tgba_emptiness_check()`.
– H^{fifo} est une table indiquant le père de chaque état visité lors du parcours en largeur. En itérant les interrogations de cette table, pour passer du père au grand-père, etc. Il est possible de trouver le chemin de plus court entre tout état visité et la racine de la recherche en largeur.
– $\mathcal{C}_k$ est une CFC au sens l'interface donnée par la classe `connected_component`. Ici, son `index` est fourni par `get_index()` et il est noté $root_{\mathcal{C}_k}$.
– Le vecteur $\langle sequence_1, ..., sequence_{n-1} \rangle$ avec n le nombre de composantes fortement connexes, donne la séquence d'états qui déterminera notre préfixe. $\forall i \in [1, n-1], sequence_i$ est la séquence (encodé sous forme de `liste`) d'états caractérisant le *chemin* traversant la composante connexe $\mathcal{C}_i$.

L'algorithme est présenté figure 4.23.

– Si $root\_component$ est composé d'une unique CFC (la CFC acceptante) alors le préfixe se résume à l'état initial. Sinon (ligne 3) un chemin traversant toutes les CFC excepté la dernière est calculé.
– *ligne 11* : les successeurs sont visités en largeur.
– Dans le cas où l'état courant appartient à la CFC suivante (ligne 13) le chemin reliant la racine de la recherche à cet état peut être construit grâce à la table H^{fifo}, et la recherche à l'intérieur de cette CFC prend fin.
– Autrement le parcours en largeur continue classiquement (ligne 30).

Ce préfixe calculé est stocké dans l'attribue `orefix` de la classe `emptiness_check`.

Théorème 3. *L'algorithme 4.23 construit une séquence d'états représentant un ensemble de préfixes possibles.*

Idée de la démonstration : Elle est immédiate et repose sur la propriété 3. Puisque $root\_component$ vérifie cette propriété, il existe un *chemin* entre l'état initial et la dernière CFC traversant toutes les composantes. Il existe un chemin entre deux CFC qui se suivent dans le parcours en profondeur de l'*emptiness check* et le parcours en largeur de notre algorithme trouve un chemin pour passer d'une composante à la suivante.

Après avoir calculé un préfixe, les méthodes `accepting_path` et `complete_cycle` permettent de construire la période d'un contre-exemple.

La méthode `accepting_path()` présentée figure 4.24 construit un chemin de la CFC passant par toutes les conditions d'acceptation, en effectuant plusieurs itérations. Chaque itération est une recherche en profondeur qui s'arrête sur les états déjà visités, c'est-à-dire qui énumère des chemins acycliques. Des chemins visités par la première recherche est retenu celui qui passe par le plus grand nombre d'ensemble d'acceptation (et s'il reste encore du choix, le plus court de ceux-ci). À partir de l'état atteint par ce premier chemin, une seconde recherche va à nouveau énumérer tous les chemins acycliques. Cependant, cette fois seuls les ensembles d'acceptation qui n'ont pas été traversés par le chemin précédent sont considérées pour le choix de celui-ci. À nouveau, une recherche peut être entreprise à partir de l'état final du précédent chemin pour chercher à traverser des ensembles d'acceptation qui n'ont pas été visités par les précédents chemins, etc. Lorsque tous les ensembles d'acceptation ont été visités, les chemins mis bout à bout constituent un chemin passant par tous ces ensembles : il ne reste plus qu'à fermer le circuit, c'est-à-dire revenir à l'état initial depuis l'état atteint par le dernier chemin. Cette dernière étape est réalisée avec un parcours en largeur dans la méthode `complete_cycle()`.

4.2.4.4 magic_search

```
class magic_search
{
public:
  magic_search(const tgba_tba_proxy *a);

  bool check();

  std::ostream& print_result(std::ostream& os,
                             const tgba* restrict = 0) const;
private:
  // ...
```

```
1  counter_example(⟨AP, Q, δ, q0, F⟩, (C1, ...Cn))
2  {
3    if (n > 1)
4      {
5        queue.push_back(⟨q0, δ(q0)⟩);
6        forall k ∈ [1, n[
7          {
8            while (queue ≠ ∅)
9              {
10               ⟨q, succ⟩ ← queue.pop_front();
11               forall q′ ∈ succ
12                 {
13                   if (q′ ∈ C_{k+1})
14                     {
15                       sequence_k.push_front(q′);
16                       sequence_k.push_front(q);
17                       q_curr ← q;
18                       root_{C_k} ← C_k.get_index();
19                       ⟨q_curr, H_{q_curr}⟩ ← H[q_curr]);
20                       while (root_{C_k} < H_{q_curr})
21                         {
22                           ⟨q_curr, q_father⟩ ← H^fifo[q_curr]);
23                           sequence_k.push_front(q_father);
24                           q_curr ← q_father;
25                           ⟨q_curr, H_{q_curr}⟩ ← H[q_curr]);
26                         }
27                       queue.clear();
28                       break;
29                     }
30                   else
31                     {
32                       if (q′ ∈ C_k ∧ q′ ∉ H^fifo)
33                         {
34                           queue.push_back(⟨q′, δ(q′)⟩);
35                           H^fifo[q′] ← q);
36                         }
37                     }
38                 }
39             }
40           q_new_start ← sequence_k.back();
41           queue.push_back(⟨q_new_start, δ(q_new_start)⟩);
42           }
43       }
44     else
45       {
46         sequence_1.push_front(q0);
47       }
48   prefix ← concat(sequence_1, ..., sequence_n);
49   return prefix;
50 }
```

FIG. 4.23 – Algorithme de construction du préfixe de contre-exemple.

```
1  accepting_path(⟨AP, Q, δ, q_0, F⟩, C_n, q_n, To_accept)
2  {
3    stack.push(⟨q_n, δ(q_n), ∅⟩);
4    not_first_time ← ⊥;
5    H^path[q_n] ← 1;
6    while (stack ≠ ∅)
7      {
8        ⟨q_step, succ_step, acc_step⟩ ← stack.pop();
9        if (succ_step = ∅)
10         {
11           H^path.erase(q_step);
12           tmp_lst.pop_back();
13         }
14       else
15         {
16           ⟨q_curr, acc_curr, prop_curr⟩ ← succ_step.remove_one();
17           tmp_lst.push_back(⟨q_curr, prop_curr⟩);
18           new_acc = acc_curr ∪ acc_step;
19           stack.push(⟨q_curr, δ(q_curr), acc_curr⟩);
20           if (q_curr ∈ C_n ∧ q_curr ∉ H^path)
21             {
22               H^path[q_curr] ← 1;
23             }
24           else
25             {
26               if (not_first_time)
27                 {
28                   if (best_acc ∩ To_accept = tmp_acc ∩ To_accept)
29                     {
30                       if (tmp_lst.size() < best_lst.size())
31                           best_lst ← tmp_lst;
32                     }
33                   else
34                     {
35                       if (tmp_acc ∩ To_accept ⊂ (best_acc ∩ To_accept))
36                           best_lst ← tmp_lst; best_acc ← tmp_acc;
37                     }
38                 }
39               else
40                 {
41                   best_lst ← tmp_lst;
42                   best_acc ← tmp_acc;
43                   not_first_time ← ⊤;
44                 }
45             }
46           stack.push(⟨q_step, succ_step, acc_step⟩)
47         }
48     }
49   if (best_acc ≠ To_accept)
50     accepting_path(⟨Σ, Q, δ, q_0, F⟩, C_n, best_lst.back().etat(), To_accept \ best_acc);
51   else if (best_lst ≠ ∅)
52     complete_cycle(⟨Σ, Q, δ, q_0, F⟩, C_n, best_lst.back().etat(), q_n);
53 }
```

FIG. 4.24 – Algorithme de construction de la période du contre-exemple.

```
};
```

La classe `magic_search` implémente l'algorithme *magic search* de Godefroid et Holzmann [33] mentionné section 2.4.3.

Cet algorithme trouve un chemin acceptant dans un automate de Büchi avec conditions d'acceptation classiques (non généralisées) portant sur les états. Les propriétés atomiques à vérifier peuvent être sur les états ou sur les transitions, car elles ne sont pas utilisées par l'algorithme.

Pour `libspot` l'algorithme manipule donc des `tgba_tba_proxy` (section 4.2.3.10).

Le *magic search* fonctionne sur le principe d'une double recherche en profondeur, mais les deux recherches partagent la même pile. La première détecte des états acceptants. Lorsque tous les successeurs d'un état acceptant ont été visités par cette première recherche, la seconde essaye de trouver un circuit autour de cet état. Autrement dit, la seconde recherche est exécutée dans l'ordre postfixe des état acceptants trouvés par la première.

Une table retenant les états visités distinguent le statut de l'état vis-à-vis des deux types de recherche : soit l'état n'a pas été visité (00), soit il a été visité uniquement par la première recherche en profondeur (10), soit il a été visité uniquement par la seconde (01), soit il a été visité par les deux (11). Ces quatre configurations sont encodées par deux bits par état dans la table. Chaque état est visité au plus en fois par chaque type de recherche, l'algorithme est donc linéaire par rapport au nombre d'états de l'automate.

Comme les propositions atomiques sont portées sur les transitions dans Spot, l'algorithme original de Godefroid et Holzmann [33] a été légèrement modifié pour retenir les formules étiquetant les transitions franchies. Il faut avouer que ce travail aurait tout aussi bien pu être fait séparément : c'est-à-dire calculer une séquence d'états acceptée, et trouver ensuite une séquence de transitions à partir de la première. En effet, dans un `tgba_tba_proxy` soit toutes les transitions d'un état sont acceptantes, soit aucune ne l'est ; une séquence d'états suffit donc à définir une famille de chemins acceptants (ce sont toutes les séquences de transitions qui passent par ces états). La figure 4.25 montre la version itérative de l'algorithme, telle que `libspot` l'implémente.

- H est une table de hachage indiquant si un état a été visité par la première recherche ($\langle q, \bot\rangle \in H$), ou par la seconde ($\langle q, \top\rangle \in H$).
- $stack$ est la pile de recherche en profondeur. Elle contient des triplets $\langle q, magic, S\rangle$. À chaque état q est associé un bit $magic$ qui indique si l'algorithme est en train d'effectuer la première recherche ($\bot$) ou la seconde ($\top$), et un ensemble S contenant tous les successeurs de q non encore explorés.
- x est le dernier état acceptant pour lequel la seconde recherche en profondeur a été lancée.
- $tstack$ est une pile parallèle à $stack$ qui indique les conditions étiquetant les transitions empruntées entre chaque état de $stack$.
- F est l'ensemble des états acceptant de l'automate.

Pratiquement L'algorithme est implémenté sous la forme d'un objet qui conserve l'état de l'algorithme (H, $stack$, $tstack$, x, ainsi que l'automate). Après chaque appel de `magic_search::check()` retournant $\top$, les piles $stack$ et $tstack$ présentent un chemin acceptant. Pour le moment ces piles ne sont pas publiques et la seule façon d'obtenir un chemin est de l'afficher avec la méthode `magic_search::print_result()`. Le paramètre facultatif `restrict` permet de projeter les états sur un automate avant l'affichage (cf. section 4.2.3.9 page 72).

La méthode `magic_search::check()` peut être appelée plusieurs fois pour énumérer d'autres chemins acceptants. Cependant, même s'il est possible de répéter cette recherche cela ne permet pas forcément de lister *tous* les chemins existants : la seule garantie est que le *magic search* exhibera au moins un chemin acceptant s'il en existe.

4.2.4.5 `tgba_reachable_iterator` et fils

Les fonctions `dotty_reachable` et `tgba_save_reachable` permettent de sauvegarder un automate, l'une dans le format de `dot` pour une représentation graphique[14], l'autre dans un format textuel qui peut ensuite être relu par la fonction `tgba_parse` afin de créer un automate `tgba_explicit`.

La logique de ces deux fonctions est similaire en ce sens que toutes deux effectuent un parcours en largeur de l'automate et effectuent des opérations chaque fois qu'un nouvel état est atteint ou une transition traversée. Ce « cœur » algorithmique a donc été abstrait dans un classe séparée (`tgba_reachable_iterator`) afin d'en favoriser la réutilisation. La figure 4.26 montre la hiérarchie obtenue.

La méthode `tgba_reachable_iterator::run()` est une méthode patron qui implémente un algorithme de recherche sur un graphe, sans présumer du sens de cette recherche. La méthode appelle `add_state()` pour déclarer les successeurs non visités de l'état courant, et appelle `next_state()` pour connaître le prochain état à visiter. Selon l'implémentation de `add_state()` et `next_state()` il est possible de créer un parcours en

[14] C'est ainsi que furent dessinés tous les automates de ce rapport.

```
magic_search(⟨AP, Q, δ, q0, F⟩)
{
   stack.push(⟨q0, ⊥, δ(q0)⟩);
   H.insert(⟨q0, ⊥⟩);
   while (stack ≠ ∅)
     {
     recurse:
       ⟨q, magic, succ⟩ ← stack.pop();
       while (succ ≠ ∅)
         {
           ⟨props, q′⟩ ← succ.remove_one();
           if (magic = ⊤ ∧ q′ = x)
             {
               tstack.push(props);
               return ⊤;
             }
           else if (⟨q′, magic⟩ ∉ H)
             {
               stack.push(⟨q, magic, succ⟩);
               tstack.push(props);
               stack.push(⟨q′, magic, δ(q′)⟩);
               H.insert(⟨q′, magic⟩);
               goto recurse;
             }
         }
       if (magic = ⊥ ∧ q ∈ F ∧ ⟨q, ⊤⟩ ∉ H)
         {
           x ← q;
           stack.push(⟨q, ⊤, δ(q′)⟩);
           H.insert(⟨q, ⊤⟩);
         }
       else if (stack ≠ ∅)
         {
           tstack.pop();
         }
     }
   return ⊥;
}
```

FIG. 4.25 – Algorithme *magic search*.

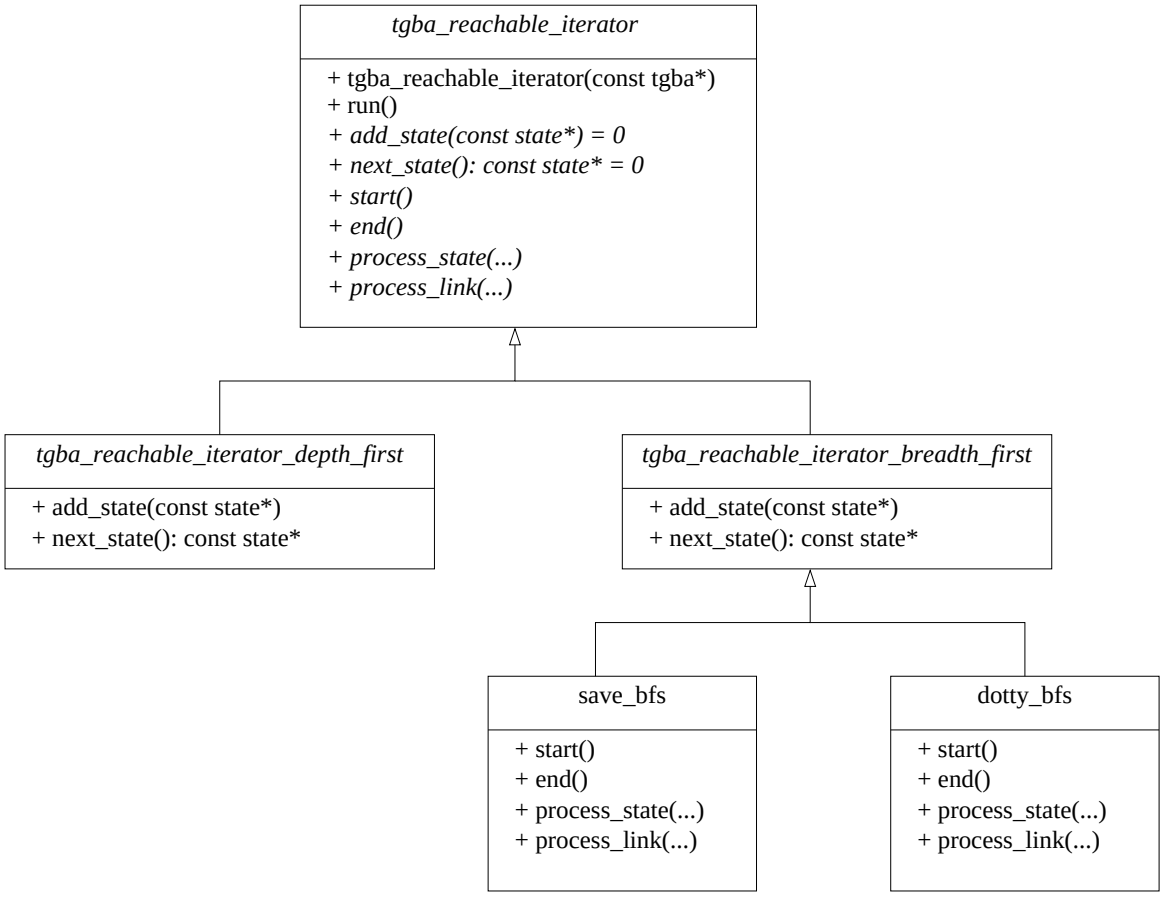


FIG. 4.26 – Hiérarchie de `tgba_reachable_iterator`.

profondeur ou en largeur, c'est ce que font les sous-classes `tgba_reachable_iterator_depth_first` et `tgba_reachable_iterator_breadth_first`.

Au cours de l'exploration de l'automate, la méthode `run()` appelle quatre autres méthodes dont l'implémentation par défaut est vide, il s'agit de `start()` au début de la recherche, `end()` à la fin, `process_state()` sur chaque nouvel état (dans l'ordre préfixe), et `process_link()` lors du franchissement d'un transition. Les classes `dotty_bfs` or `save_bfs` définissent uniquement ces quatre fonctions pour effectuer leur sorties.

Les fonctions `dotty_reachable` et `tgba_save_reachable` ne sont que des façades dissimulant l'emploi des classes `dotty_bfs` ou `save_bfs`.

4.2.4.6 `lbtt_reachable`

```
std::ostream& lbtt_reachable(std::ostream& os, const tgba* g);
```

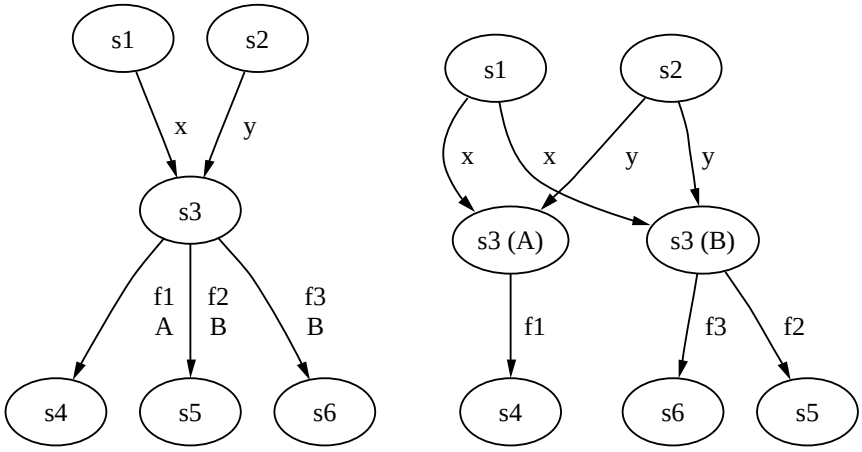
`lbtt_reachable()` affiche un automate au format LBTT [64].

La difficulté de cette fonction tient à ce que LBTT ne manipule pas les mêmes automates que Spot. LBTT gère des automates de Büchi généralisés dans lesquels les propositions atomiques sont portées par les transitions, et les conditions d'acceptation par les *états*.

La conversion d'un `tgba` en un automate pour LBTT consiste donc à déplacer les conditions d'acceptation des transitions vers les états.

Le choix fait ici est de dupliquer un état autant de fois qu'il existe de conditions d'acceptation différentes sur les transitions sortantes. La figure 4.27 illustre l'opération sur un état (S3). Toutes les transitions entrantes sont naturellement aussi dupliquées.

Cette opération est reproduite sur tous les états de l'automate. Si l'automate possède $|Acc|$ ensembles d'acceptation, il existe $2^{|Acc|}$ conditions d'acceptation possibles. Dans le pire cas, lorsque chaque état de l'automate possède des transitions sortantes étiquetées par chacune des $2^{|Acc|}$ conditions d'acceptation, le nombre d'états et de transitions de l'automate sera donc multiplié par $2^{|Acc|}$.

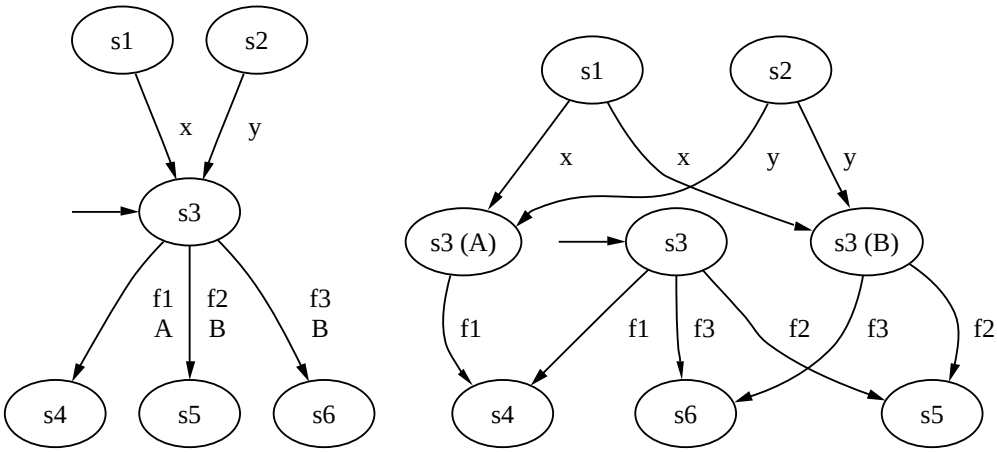


(a) (b)

FIG. 4.27 – L'état s_3 est dupliqué afin de déplacer les conditions d'acceptation des transitions vers les états.

Enfin, LBTT (comme Spot) n'accepte qu'un seul état initial dans ses automates. Un problème se pose donc lorsque l'état initial a dû être dupliqué. Par exemple si l'état s_3 de la figure 4.27(a) est un état initial, aucun des deux états correspondants de la figure 4.27(b) ne peut servir *seul* d'état initial.

Dans ce cas, `lbtt_reachable()` ajoute une nouvelle copie de cet état, sans aucune transition entrante et avec toutes les transitions sortantes d'origine. Ce nouvel état sert uniquement d'état initial, et comme il ne peut être atteint, sa condition d'acceptation importe peu. La figure 4.28, reprend l'exemple précédent dans le cas où s_3 est un état initial.



(a) (b)

FIG. 4.28 – Cas où s_3 est un état initial.

Une autre façon de convertir un automate sous une forme utilisable par LBTT est de le dégénéraliser à l'aide d'un `tgba_tba_proxy`. Cette conversion a l'avantage de multiplier la taille de l'automate au pire par $1+\max(|Acc|,1)$, c'est-à-dire qu'elle est linéaire par rapport à $|Acc|$ plutôt qu'exponentielle.

L'approche choisie pour `lbtt_reachable()` peut se justifier vis-à-vis de `tgba_tba_proxy` de plusieurs façons.

– Historiquement, `lbtt_reachable()` a été implémentée avant `tgba_tba_proxy`.
– Les deux approches ne sont pas incompatibles. Comme toutes les transitions sortantes des états d'un automate `tgba_bdd_proxy` possèdent la même condition d'acceptation, `lbtt_reachable()` ne ferra aucune duplication d'état sur un tel automate.

– Malgré un pire cas plus grand, les `tgba` convertis par `lbtt_reachable()` semblent en moyenne plus petits que ceux produits par `tgba_tba_proxy` en moyenne. Ce point est évoqué section 5.2.

4.2.5 Interface avec GreatSPN

GreatSPN est un outil d'analyse de réseaux de Petri bien formés [13]. Parmi les opérations que cet outil peut effectuer sur ces réseaux, nous nous intéressons à l'exploration du graphe des marquages accessibles. GreatSPN possède différents modes de réduction du graphe des marquages fondés sur la détection des symétries de comportement [11, 12, 37, 17, 36, 35].

Un graphe des marquages accessibles est une structure de Kripke, aussi, comme indiqué section 3.3, il peut être vu comme un `tgba`. L'un des objectifs du stage était donc d'interfacer Spot et GreatSPN pour effectuer du *model checking* sur des réseaux de Petri bien formés, et bénéficier des réductions de l'espace d'état effectuées par GreatSPN.

GreatSPN étant un programme, Yann THIERRY-MIEG et Souheib BAARIR actuellement en thèse au Lip6 et travaillant sur ces symétries, ont modifié GreatSPN pour en faire une bibliothèque et nous offrir une interface en C. Cette interface contient les fonctions suivantes.

```
int initialize(int argc, char** argv);
int finalize(void);

int print_state(const State s, char** st);

int prop_index(const char* name, AtomicProp* propindex);
int prop_kind(const AtomicProp prop, AtomicPropKind* kind);

int initial_state(State *M0);

int satisfy(const State s, const AtomicProp props[],
            unsigned char** truth, size_t props_size);
int satisfy_free(unsigned char* truth);

typedef struct EventPropSucc {
  State s;
  AtomicProp p;
} EventPropSucc;

int succ(const State s, EventPropSucc** succ, size_t* succ_size);
int succ_free(EventPropSucc* succ);
```

Les différents de types réductions du graphe des marquages accessibles peuvent être choisis en liant Spot avec différentes instances[15] de la bibliothèque GreatSPN. L'interface reste la même dans tous les cas.

Les deux premières fonctions permettent d'initialiser GreatSPN. Les paramètres d'`initialize()` indiquent les fichiers utilisés par GreatSPN pour représenter son modèle. Ce modèle fait intervenir des propriétés, et ces propriétés apparaîtrons dans Spot sous forme de propositions atomiques dans les formules. GreatSPN indice chaque propriété par un entier, et la fonction `prop_index()` permet de convertir en une chaîne désignant la propriété en son indice.

Pour la suite, nous distinguons deux types de propositions : les propositions qui ont trait aux états du système, et les propositions dites *événementielles* qui ont trait aux transitions franchies. La fonction `prop_kind()` permet de distinguer ces deux types.

Ensuite nous retrouvons les fonctions qui permettent de parcourir le graphe. `initial_state()` retourne l'état initial, `satisfy()` permet d'obtenir les propriétés qui étiquettent les états, et `succ()` permet de parcourir les successeurs d'un état. Comme les transitions peuvent être étiquetées par au plus une propriété événementielle, les successeurs sont des paires (*proposition, état*).

Un modèle peut posséder énormément de propriétés, et elles ne seront pas toutes utilisées lors du *model checking* avec une formule LTL. Évaluer la satisfaction de chacune de ces propriétés prend du temps. Aussi, pour limiter les calculs de GreatSPN, la fonction `satisfy()` permet de préciser la liste des conditions qui sont nécessaires à la formule à tester : c'est-à-dire celles qui sont nécessaires dans l'état courant lors du produit synchronisé. Les propositions d'intérêt sont indiquées à l'aide de l'argument `props`.

[15] C'est-à-dire qu'il existe plusieurs versions de la bibliothèque GreatSPN, compilées avec différentes options. Ce choix empêche la sélection dynamique de l'algorithme de réduction utilisé, mais il est dicté par l'organisation du code de GreatSPN.

Au niveau de Spot, l'interface avec GreatSPN est implémentée comme une bibliothèque séparée[16] (figure 4.1) qui fournit les objets `gspn_environment` (sous-classe de la classe `ltl::environment`, présentée section 4.2.2.1), et `tgba_gspn` (sous-classe de `tgba`, cf. section 4.2.3.5).

Ces deux classes utilisent l'interface C de `gspn` pour réaliser toutes les opérations. Toutes les propositions doivent être déclarées auprès de l'instance de `gspn_environment` avant d'être utilisées (par exemple avant de pouvoir construire une formule LTL avec `ltl::parse()`).

Lors de l'exploration de l'automate, l'itérateur sur les successeurs créé par `tgba_gspn::succ_iter()` utilise `satisfy()` pour obtenir les propositions de l'état courant, et `succ()` pour obtenir la liste des successeurs et les propositions événementielles. Toutes ces propositions sont ensuite transformées en BDD et retournées sur chaque transition au fur et à mesure de l'itération.

Pour optimiser les produits, la méthode `compute_support_conditions()` (section 4.2.3.9, page 71) est implémentée de façon à retourner toutes les conditions qui étiquettent l'état courant. Ceci ignore les conditions qui étiquettent les transitions.

Sur ce même principe, il est prévu d'utiliser la méthode `support_variables()`, pour obtenir l'ensemble des variables utilisées localement lors du calcul des successeurs d'un état du produit, et passer cette information à GreatSPN pour limiter ces calculs. Ceci reste à faire.[17]

Grâce à cette interface sous forme d'objets `gspn_environment` et `tgba_gspn`, GreatSPN peut être utilisé avec tous les algorithmes de `libspot` au même titre que n'importe quel `tgba`. D'autre part, le fait que cette intégration a pris la forme d'une bibliothèque séparée démontre l'extensibilité de `libspot` *de l'extérieur*.

Il faut cependant nuancer ce dernier point. La méthode `support_variables()` n'a été introduite dans la classe `tgba` et toutes ses sous-classes uniquement parce que nous savions qu'elle servirait dans `tgba_gspn`. Naturellement, il aurait été possible de s'en passer (c'est d'ailleurs le cas actuel, puisqu'elle n'est pas encore utilisée). Mais sans elle il serait impossible de limiter les calculs que GreatSPN doit effectuer. En conclusion, `libspot` peut certes être étendue de façon externe, mais les optimisations de ces modules externes peuvent nécessiter un travail à l'intérieur de la bibliothèque.

4.2.6 Interface avec Python

Python[18] est un langage de script extensible, avec lequel nous avons voulu interfacer `libspot` pour faciliter les expérimentations.

Le langage Python est interprété, et typé dynamiquement. Ces deux points sont souvent pris pour synonymes de *lent*, mais lorsqu'il s'agit de fournir uniquement une interface de haut niveau le problème ne se pose pas. La couche ajoutée par Python est insignifiante comparée au temps occupé par les algorithmes de `libspot`.

L'intérêt de l'interface avec un tel langage est de permettre de réaliser des tests interactivement, de débugger, ou de prototyper des applications plus rapidement qu'avec un langage tel que C++ (dans lequel le cycle écriture–compilation–exécution est très handicapé par l'étape intermédiaire).

La façon d'interfacer une bibliothèque C ou C++ avec Python est de créer une fonction « façade » pour chaque fonction publique de la bibliothèque. Le rôle de cette façade est de traduire chaque paramètre de la fonction depuis sa représentation en Python vers son équivalent C ou C++, d'appeler la véritable fonction, puis et de faire l'opération inverse pour convertir la valeur de retour. De similaires façades peuvent être implémentées pour les variables globales, les objets, et leurs méthodes.

Plutôt que de programmer toutes ces façades à la main, Spot utilise SWIG[19] pour les générer automatiquement. SWIG est un outil qui lit la déclaration d'une interface en C ou C++, puis peut générer automatiquement des façades pour divers langages de scripts (Perl, Python, Tcl/Tk, Guile, ...) On pourrait imaginer supporter tous ces langages dans Spot, mais Python est bien suffisant pour le moment.

SWIG lit l'interface directement sous la forme d'un fichier d'en-tête (`*.h`), mais les définitions doivent être annotées pour tirer meilleur parti de l'outil. Par exemple une fonction qui alloue l'objet qu'elle retourne doit être explicitement déclarée comme telle afin que l'objet soit détruit au moment opportun (par le ramasse miette de Python).

Spot contient deux modules Python ainsi traduits : `buddy.py` et `spot.py`. Ces deux modules sont séparés parce que d'une part techniquement ils correspondent à des bibliothèques différentes, d'autre part nous avons été surpris de ne trouver aucune interface Python déjà existante pour BuDDy et espérons que celle-ci pourra être utilisée indépendamment de Spot.

[16] Deux raisons à cela. Une raison philosophique : conserver l'indépendance de `libspot`. Une raison technique : la version « bibliothèque » de GreatSPN dont nous disposons n'est qu'une bibliothèque statique, dont `libspot` (qui est dynamique) ne peut dépendre.

[17] La machinerie est en place, il reste juste à appeler les méthodes au bon moment et à les tester.

[18] www.python.org

[19] www.swig.org

L'annexe A présente un exemple de programme utilisant BuDDy écrit une fois en C++ et une fois en Python, à titre d'illustration. Il est regrettable que nous n'ayons réalisé cette interface avec Python que vers la fin de stage, car elle aurait été très utile au début, alors que nous tâtonnions avec BuDDy. L'utilisation de l'interpréteur interactif de Python, pour appeler les fonctions de BuDDy directement et visualiser les résultats aurait été très pratique pour débuter.

L'ensemble des objets et méthodes publiques de Spot est disponible depuis Python via le module `spot.py`. À l'aide de ce module, et en utilisant les facilités de la bibliothèque standard de Python pour écrire des script CGI, nous avons développé un script qui permet de traduire une formule LTL en un automate avec `ltl_to_tgba_lacim()` (section 4.2.4.1) ou `ltl_to_tgba_fm()` (section 4.2.4.2) sur le Web.

À part ce script et divers programmes de tests, aucune autre application n'utilise encore l'interface Python. L'objectif, à terme, est de fournir un outil en ligne de commande, écrit en Python, pour vérifier une formule LTL sur un réseau de Petri en passant par GreatSPN. Il faudra pour cela écrire la version Python de `tgba_gspn` et `gspn_environment`[20].

[20] Une chose impossible actuellement parce que les bibliothèques de GreatSPN ne sont pas dynamiques.

Chapitre 5

Évaluation

5.1 Batterie de tests

La bibliothèque possède une petite batterie de tests complétée au fur et à mesure des développements de la bibliothèque. Il n'y a pas grand chose à en dire, si ce n'est qu'elle existe, et permet de considérer des changements importants avec plus de sérénité : la batterie sert de garde fou.

Ces tests sont actuellement classés en quatre catégories pour le module LTL, le module TGBA, l'interface avec GreatSPN, et celle avec Python.

Parmi les tests, on en trouve plusieurs qui se contentent d'appeler des algorithmes sans vérifier leur sortie. Bizarrement, de tels tests sont assez utiles. D'abord ils fournissent un cas d'utilisation de la fonctionnalité, et en quelque sorte s'assurent que celle-ci peut toujours s'utiliser de la même façon. Ensuite ces tests font tourner le code de `libspot` ce qui peut déclencher des assertions à plusieurs endroits.

5.2 Interface avec LBTT

LBTT, développé par Heikki Tauriainen [62, 64], est un environment de test de traducteurs de formule LTL en automate de Büchi.

Nous avons déjà mentionné certains des tests qu'il effectue section 2.7, et si nous avons présenté la traduction d'un `tgba` en automate compréhensible par LBTT section 4.2.4.6, c'est que cet outil est utilisé dans Spot pour tester les deux traducteurs (`ltl_to_tgba_lacim()` section 4.2.4.1, et `ltl_to_tgba_fm()` section 4.2.4.2), ainsi que l'opération de dégénéralisation (`tgba_tba_proxy`, section 4.2.3.10).

L'interface a été un peu difficile à réaliser parce que LBTT est une espèce de boite noire sur laquelle il faut connecter le programme de traduction à tester, sans accès aisé aux données qui lui sont remises (la formule) et vice-versa aux données qu'il renvoie. Par exemple LBTT contient un algorithme très paramétré pour générer des formules, mais ne fournit pas de moyen explicite pour générer une telle formule. La seule façon de provoquer cette génération est de lancer un test du programme de traduction. Bien sûr pour développer une interface de ce programme de traduction qui réponde aux exigences de LBTT, il aurait été utile de disposer d'une ou deux formules telles qu'elles sont produites par LBTT...

Quoiqu'il en soit, après quelques tâtonnements et quelques modifications de LBTT pour en faciliter l'utilisation dans la batterie de tests de Spot, l'outil est une vraie merveille. Il a permis de trouver quelques problèmes dans les premières implémentations des différents algorithmes. Les problèmes trouvés n'étaient pas tous liés à l'algorithme en lui-même, mais certaines formules générées automatiquement provoquaient des situations (telles que la mauvaise gestion d'un compteur de références dans les dictionnaires ou les formules) qui déclenchaient l'un des `assert()` encombrant le code de `libspot`.

L'utilisation de LBTT dans la batterie de tests de spot est la suivante : LBTT effectue l'ensemble de ses vérifications sur une base de 100 formules générées automatiquement, sur les quatre traducteurs suivants[1] :

- Couvreur/LaCIM (i.e., `ltl_to_tgba_lacim()`)
- Couvreur/LaCIM dégénéralisé (i.e., `ltl_to_tgba_lacim()` + `tgba_tba_proxy`)
- Couvreur/FM (i.e., `ltl_to_tgba_fm()`)
- Couvreur/FM dégénéralisé (i.e., `ltl_to_tgba_fm()` + `tgba_tba_proxy`)

[1] Les tests de LBTT incluent aussi des vérifications inter-traductions

L'exécution de ce test prend un quart d'heure sur une machine à 2GHz.

Outre ces tests, LBTT fournit aussi des statistiques sur l'ensemble des automates utilisés pendant son exécution. Une copie de ces statistiques est fournie en annexe B. Malheureusement nous avons déjà expliqué (section 4.2.4.6) que la taille des automates manipulés par LBTT n'était pas comparable avec celle des automates `tgba`.

Il est donc difficile de comparer les deux algorithmes au regard de ces statistiques. Cependant, il est quand même intrigant de comparer les tailles moyennes des 100 automates produits par chaque traduction.

	états	transitions	ens. d'acceptation
Couvreur/LaCIM	10.40	61.80	0.78
Couvreur/LaCIM dégénéralisé	15.21	66.89	0.98
Couvreur/FM	10.53	62.70	0.64
Couvreur/FM dégénéralisé	8.86	33.06	0.98

Rappelons que l'opération `lbtt_reachable()` sur un automate dégénéralisé ne produit aucun changement dans l'automate ; seuls les automates généralisés peuvent souffrir des duplications d'état (cf. section 4.2.4.6). Les statistiques des versions généralisées et dégénéralisées des traducteurs permettent donc de comparer deux façons de convertir un `tgba` en un automate dont les conditions d'acceptation portent sur les états au lieu des transitions : la méthode utilisée par `lbtt_reachable()` ou une dégénéralisation avec `tgba_tba_proxy`.

Il est amusant de constater que `lbtt_reachable()` semble plus efficace que `tgba_tba_proxy` sur les automates produits par `ltl_to_tgba_lacim()`, alors que c'est l'inverse pour ceux issus de `ltl_to_tgba_fm()`.

5.3 Directions futures

Nous n'avons malheureusement pas eu le temps de comparer Spot à d'autres outils. Qu'il s'agisse de *model checkers* complets, ou simplement des traducteurs de formule LTL. C'est une étape importante, et des résultats trop décevants pourraient amener à reconsidérer des choix tels que l'emploi de classes abstraites pour stocker les états et l'utilisation de BDD comme représentation homogène des étiquettes des transitions.

L'interface des différents types d'*emptiness check* devrait être retravaillée et unifiée.

Les dictionnaires devront être retravaillés pour s'approcher de l'architecture à deux niveaux décrite section 4.2.3.1. Ceci devrait entre autre permettre d'ajouter un nouveau type d'automate utilisant les BDD[2], mais considérant les états comme les éléments d'un ensemble d'états et non comme des conjonctions de Now[·] et Next[·]. Ceci est plus intéressant car, la représentation d'un élément d'un ensemble de taille N demande seulement $\log_2 N$ variables.

Une conversion d'un automate `tgba` quelconque vers ce nouveau type d'automate devra être prévue.

L'encodage des conditions d'acceptation gagnerait à être retravaillé pour utiliser une représentation ensembliste classique. L'encodage utilisé actuellement était nécessaire par le passé lorsque les conditions d'acceptation devaient être traduites au passage d'un dictionnaire à un autre. Comme les dictionnaires sont maintenant partagés entre les automates, ces traductions n'existent plus et la représentation pourra être améliorée.

Couvreur [20] donne deux optimisations qui peuvent améliorer son algorithme de traduction. Il faudra les implémenter.

Il faudra implémenter un système de réécriture dans le module LTL, afin de pouvoir simplifier les formules aisément avec des règles telles que celles décrites section 2.3.2. Peut-être en utilisant une interface de *matching* comparable à celle développée par van den Brand et al. [67].

De la même façon, il serait bon d'implémenter des post-traitements tels que ceux décrits section 2.3.3.

5.4 Évolution du code

À titre informatif, la figure 5.1 représente l'évolution de la taille de Spot pendant le stage. Les différentes courbes présentées correspondent au module LTL (`src/ltl*/`), au module TGBA (`src/tgba*/`), au module GreatSPN (`iface/`), au module Python (`wrap/`), et à diverse routines orphelines (`src/misc/`).

Comme cela se voit le développement à d'abord concerné le module LTL. Une mini interface en Python de ce module a été faite le 30 avril pour confirmer le choix de SWIG. Le 15 mai, la hiérarchie des `formula` a été récrite de façon à partager les sous-formules identiques. Viennent ensuite le développement du module TGBA. Le trou du 14 juillet

[2] La prévision de l'ajout d'autre forme d'automate reposant sur les BDD est la raison pour laquelle la classe `tgba_bdd_concrete` ne s'appelle pas simplement `tgba_bdd`. Il n'y a par contre aucune justification à l'emploi du terme `_concrete` plutôt qu'un autre.

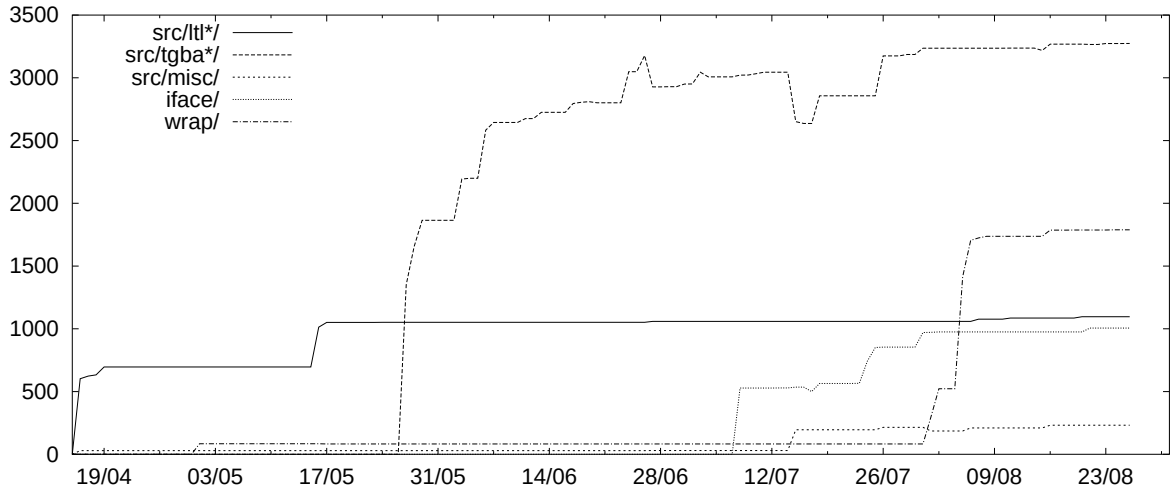


FIG. 5.1 – Évolution du nombre de lignes des différents modules au cours du stage.

correspond au changement de politique de l'utilisation des dictionnaires décrit section 4.2.3.1. Enfin, les interfaces avec GreatSPN et Python ont été réalisées. Le bond du 5 août correspond à l'ajout du script CGI de traduction LTL.

Chapitre 6

Conclusion

Nous avons présenté Spot, une bibliothèque de *model checking* orientée objets développée en C++ au cours des six mois de ce stage. Ces principales caractéristiques sont les suivantes.

- Spot manipule des automates des Büchi étiquetés sur les transitions (TGBA).
  Ces automates permettent de représenter des formules LTL de façon plus compacte que ceux étiquetés sur les états. Ils permettent aussi de représenter des automates étiquetés sur les états, ce qui autorise l'utilisation d'automates produits par d'autres outils. En revanche ils sont incompatibles avec les algorithmes écrits pour les automates étiquetés sur les états.
  L'intérêt pour ce type d'automate étant récent, les algorithmes existants pour ces automates sont moins nombreux, et leur développement est d'actualité. Par exemple l'auteur de LBTT souhaite étendre son outil à ce type d'automates, mais ne peut le faire sans développer une nouvelle architecture et de nouveaux algorithmes[1].
  En proposant une implémentation de ces automates sous la forme d'une bibliothèque, nous espérons que Spot puisse servir de berceau au développement de nouveaux algorithmes.
- L'extensibilité de la bibliothèque est assurée par l'utilisation d'objets et de certains *design patterns*.
  Par exemple de nouveaux algorithmes manipulant des formules peuvent être écrits facilement sous la forme de visiteurs, éventuellement en réutilisant les comportements de visiteurs existants.
  Les TGBA servent de représentation pivot dans la bibliothèque et peuvent servir à représenter aussi bien des structures de donnée ou des algorithmes. Interpréter une structure de donnée (par exemple une structure de Kripke, ou un automate étiqueté sur les états) comme un TGBA permet d'utiliser tous les algorithmes attendant des TGBA. Présenter un algorithme comme un TGBA permet de réaliser des calculs à la volée, en fonction des parties explorées du TGBA ; c'est pour cela que le produit de deux automates est une classe de TGBA.
  L'interface avec GreatSPN est un bon exemple d'extension par sous-classage dans lequel l'automate est calculé à la volée.
- Les principaux algorithmes implémentés sont les suivants :
  - Deux algorithmes de traduction de formules LTL en TGBA : tous deux publiés par Couvreur [20, 21].
  - Deux algorithmes d'*emptiness check* : celui de Couvreur [20], et le *magic search* de Godefroid et Holzmann [33].
  - Une extension de l'*emptiness check* de Couvreur pour calculer un contre-exemple.
  - Un produit synchronisé à la volée, et optimisé à l'aide d'un filtre sur les successeurs.
  - Une dégénéralisation à la volée.
- Ces algorithmes sont les ingrédients d'un *model checker*. D'ores et déjà, il est possible de vérifier une formule LTL sur un réseau de Petri bien formé de la façon suivante :
  1. L'interface avec GreatSPN est utilisée pour transformer le réseau de Petri en un TGBA à la volée.
  2. L'un des deux algorithmes de traduction est utilisé pour construire un TGBA à partir de la formule.
  3. Le produit à la volée de ces deux automates est instancié sous la forme d'un troisième TGBA.
  4. Un algorithme *emptiness check* est appliqué sur ce dernier automate.
- L'interface avec Python permet de tester les combinaisons de ces différents éléments facilement.
- L'utilisation d'une batterie de tests, ainsi que celle de LBTT, assure une certaine qualité à la bibliothèque, qui d'autre part a été développée avec l'espoir d'en faire un logiciel libre (le fait qu'un source puisse être rendu publique impose aussi un certain degré de documentation, dans le code et à côté).

Il serait malhonnête de ne pas aborder les points négatifs.

- Vis-à-vis du sujet du stage, l'objectif d'évaluation des différentes méthodes et technique n'a pas été atteint. C'est notre conviction que la bibliothèque *permet* ces évaluations (au moins celles des algorithmes qui sont implémentés...) mais elles n'ont pas été conduites par manque de temps.

[1] Correspondance privée avec l'auteur.

– L'évaluation de la bibliothèque en elle-même manque. Nous avons montré que Spot permettait de construire un *model checker*, mais nous ne pouvons pas prétendre que celui-ci serait efficace, car aucune comparaison avec d'autres *model checkers* n'a été entreprise.
  Nous pensons que la combinaison des réductions de GreatSPN, de l'utilisation des TGBA, et des simplifications pendant le produit synchronisé doit se traduire par un gain substantiel par rapport aux autres approches utilisées actuellement ; mais nous ne pouvons pas encore étayer cette affirmation.

Nous comptons réaliser ces comparaisons dans un futur proche.

Bibliographie

[1] Alfred V. Aho, John E. Hopcroft, et Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1974. 2.4.1

[2] Henrik Reif Andersen. An introduction to binary diagrams. Lecture notes, October 1997. 2.6.2.1, A

[3] 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. 2.1.7

[4] Evert Willem Beth. *La crise de la raison et la logique*. Gauthier-Villars, 1957. 6

[5] Evert Willem Beth. *The Foundations of Mathematics*. North Holland, 1959. Second edition revised in 1965. 6

[6] Roderick Bloem. *Search Techniques and Automata for Symbolic Model Checking*. PhD thesis, University of Colorado, 2001. 2.3.1.6, 2.3.1.6, 2.3.2, 2.3.3.1

[7] Roderick Bloem, Harold N. Gabow, et Fabio Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In J. W. O'Leary M. D. Aagaard, editor, *Formal Methods in Computer Aided Design (FMCAD'00)*, number 2517 in Lectures Notes in Computer Science, pages 37–54. Springer-Verlag, 2000. 2.4.1

[8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, August 1986. 2.6.2, 2.6.2.1

[9] 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 [46]. 2.1.5

[10] 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. 2.6.2, 2.6.2.3

[11] 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. 4.2.5

[12] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, et Serge Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11) :1343–1360, November 1993. 4.2.5

[13] Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, et Marina Ribaudo. GreatSPN 1.7 : GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1–2) :47–68, 1995. 1.2, 4.2.5

[14] Gianfranco Ciardo, Gerarld Lüttgen, et Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In Mogens Nielsen et Dan Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN'00)*, volume 1825 of *Lectures Notes in Computer Science*, pages 103–122. Springer-Verlag, 2000. 2.6.2, 2.6.2.3

[15] Edmund M. Clarke, Orna Grumberg, et Doron A. Peled. *Model Checking*. The MIT Press, 2000. 2.2, 2.6.2.1, 2.6.2.3

[16] Edmund M. Clarke et Jeannette M. Wing. Formal methods : State of the art and future. *ACM Computing Surveys*, 28(4) :626–643, 1996. 1.1

[17] Flavio Corradini et Paola Inverardi, editors. *Exploiting Partial Symmetries for Markov Chain Aggregation*, volume 39 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, december 2000. 4.2.5

[18] Olivier Coudert. Two-level logic minimization : an overview. *Integration, the VLSI journal*, 17(2) :97–140, October 1994. 2.3.1.6

[19] 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. 2.4.3, 2.6.1.1

[20] Jean-Michel Couvreur. On-the-fly verification of 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. 1.2, 2.3.1.5, 2.3.1.5, 2.13, 2.3.1.6, 2.3.1.8, 2.3.2, 2.4.2, 2.5, 4.2.3.10, 4.2.4.2, 4.2.4.2, 4.2.4.3, 4.20, 5.3, 6

[21] 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. Université du Québec à Montréal, August 2000. 2.3.1.1, 2.3.1.6, 2.3.1.9, 2.5, 2.3.1.9, 4.2.4.1, 4.2.4.1, 6

[22] Marco Daniele, Fausto Giunchiglia, et Moshe Y. Vardi. Improved automata generation for Linear Temporal Logic. Technical report, ITC-IRSC, 1999. Later republished as [23]. 2.3.1.4

[23] 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. 2.3.1.4, 2.7, 6

[24] 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 *Lectures Notes in Computer Science*, pages 183–186. Springer-Verlag, 2001. 1.1

[25] Kousha Etessami et Gerard J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (Concur'2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2000. 2.3.1.8, 2.3.2, 2.3.3, 4, 2.8

[26] Melving Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 2nd edition, 1996. 2.1.7

[27] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995. 4.2.2.4, 4.2.3.3, 4.2.3.5

[28] 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. Springer-Verlag, 2001. 2.3.1.7, 2.3.1.7, 2.3.3, 3, 2.8

[29] 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*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall. 2.3.1.3, 6

[30] Dimitra Giannakopoulou et Flavio Lerda. Efficient translation of LTL formulae into Büchi automata. Technical Report 01.29, Research Institute for Advanced Computer Science, June 2001. Later republished as [31]. 2.3.1.8, 3

[31] Dimitra Giannakopoulou et Flavio Lerda. From states to transitions : Improving translation of LTL formulae 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. 2.3.1.8, 3, 6

[32] Patrice Godefroid, Gerard Holzmann, et Didier Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3) :227–241, Nov 1995. 2.6.1.2, 2.6.3

[33] 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. 2.4.3, 2.5, 4.2.4.4, 4.2.4.4, 6

[34] Erich Grädel, Wolfgang Thomas, et Thomas Wilke. *Automata, logic, and infinite games*, volume 2500 of *Lectures Notes in Computer Science*. Springer-Verlag, 2002. 2.1.5

[35] Serge Haddad et Jean-Michel Ilié. *Symétries et logique temporelle*, chapter 4. Traité IC2, série Informatique et systèmes d'information. Michel Diaz, février 2003. 2.6.3, 4.2.5

[36] 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. 2.6.3, 4.2.5

[37] Serge Haddad, Jean-Michel Ilié, Mohamed Taghelit, et Belhassen Zouari. Symbolic marking graph and partial symmetries. In *Proceedings of 16th International Conference on Application and Theory of Petri Nets (ICATPN'95)*, pages 238–257, Torino, Italy, 1995. 4.2.5

[38] 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, Jan 2003. 2.3.1.5

[39] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on software Engineering*, 23(5) :279–295, May 1997. 2.8

[40] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, Nov 1998. 2.6.1.1

[41] 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. 2.8

[42] Yonit Kesten, Zohar Manna, Hugh McGuire, et Amir Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubertis, editor, *Proceedings for the 5th Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lectures Notes in Computer Science*, pages 97–109. Springer-Verlag, 1993. 2.3.1.2

[43] Orna Kupferman et Moshe Y. Vardi. Weak alternating automata are not that weak. In *Proceedings of the 5st Israeli Symposium on Theory of Computing and Systems (ISTC'97)*, pages 147–158. IEEE Computer Society Press, 1997. 2.2, 2.3.1.7

[44] Orna Kupferman et Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proceedings 30th ACM Symposium on Theory of Computing*, pages 224–233. ACM, 1998. 2.2

[45] Orna Kupferman et Moshe Y. Vardi. Model checking of safety properties. In N. Halbwachs et D. Peled, editors, *Proceedinfs of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, 1999. 2.7

[46] Sounders Mac Lane et Dirk Siefkes, editors. *The Collected Works of J. Richard Büchi*. Springer-Verlag, 1990. 6

[47] François Laroussinie, Nicolas Markey, et Philippe Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th IEEE Sumposium on Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, July 2002. IEEE Computer Society Press. 2.1.6

[48] Jørn Lind-Nielsen. BuDDy : Binary Decision Diagram package. Release 2.2, November 2002. 4.2.1

[49] Zohar Manna et Amir Pnueli. *Temporal Verification of Reactive Systems – Safety*. Springer-Verlag, 1995. 2.3.1.2

[50] Nicolas Markey. Past is for free : on the complexity of verifying linear temporal properties with past. In Uwe Nestmann et Prakash Panangaden, editors, *Proceedings of the 9th International Workshop on Expressiveness in Concurrency (EXPRESS'02)*, volume 68.2 of *Electronic Notes in Theoretical of Computer Science*. Elsevier Science Publishers, 2002. 2.1.6

[51] 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 *Lectures Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001. 1.1

[52] David E. Muller, Ahmed Saoudi, et Paul E. Shupp. Alternating automata, the weak monadic theory of the tree and its complexiy. In Laurent Kott, editor, *Proceedings 13th of the International Colloquium on Automata, Languages and Programming (ICALP'86)*, volume 226 of *Lectures Notes in Computer Science*, pages 233–244. Springer, 1986. 2.2

[53] 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 *Lectures Notes in Computer Science*, pages 100–107. Springer, 1984. 2.2

[54] Enric Pastor et Jordi Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 790–795, Paris, March 1998. 2.6.2

[55] Enric Pastor, Jordi Cortadella, et Oriol Roig. Symbolic analysis of bounded Petri nets. *IEEE Transactions on Computers*, 50(5) :432–448, May 2001. 2.6.2

[56] Kavita Ravi, Roderick Bloem, et Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In J. W. O'Leary M. D. Aagaard, editor, *Proceedings of the 4th International Conference on Formal Methods in Computer Aided Design (FMCAD'00)*, volume 2517 of *Lectures Notes in Computer Science*, pages 143–160. Springer-Verlag, 2000. 2.4.1

[57] Mauno Rönkkö. LBT : LTL to Büchi conversion. http://www.tcs.hut.fi/Software/maria/tools/lbt/, 1999. Implements [29]. 2.3.1.3

[58] Silicon Graphics Computer Systems, Inc. *Standard Template Library Programmer's Guide*. 5

[59] Raymon M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. 6

[60] Fabio Somenzi et Roderick Bloem. Efficient Büchi automata for LTL formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263. Springer-Verlag, 2000. 2.3.1.6, 2.3.1.8, 2.3.2, 2.3.3, 2.3.3.1

[61] Alex Stepanov et Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1995. 5

[62] 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. 2.7, 2.7, 5.2

[63] 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/September 2000. Springer-Verlag. 2.7

[64] 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. 2.7, 4.2.4.6, 5.2

[65] Wolfgang Thomas. Languages, automata, and logic. Technical Report 9607, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Germany, May 1996. 2.1.5

[66] Antti Valmari. The state explosion problem. In W. Reisig et G. Rozenberg, editors, *Lectures on Petri Nets 1 : Basic Models*, volume 1491 of *Lectures Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998. 1.1, 2.6.3

[67] Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint, et Pieter A Olivier. Efficient annotated terms. Technical Report SEN-E0003, Centrum voor Wiskunde en Informatica, February 2002. 5.3

[68] 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. 2.1.8, 2.1.8, 2.5

[69] Moshe Y. Vardi. Alternating automata and program verification. In Jan van leeuwen, editor, *Computer Science Today –Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1995. 2.2

[70] 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*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996. 2.1.5, 2.1.8, 2.2, 2

[71] 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. 1.1

[72] Moshe Y. Vardi et Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115 (1) :1–37, Nov 1994. 6

[73] Kimmo Varpaaniemi, Keijo Heljanko, et Johan Lilius. PROD 3.2 : An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 472–475. Springer-Verlag, 1997. 2.6.3

[74] Chao Wang, Roderick Bloem, Gary D. Hachtel, Kavita Ravi, et Fabio Somenzi. Divide and compose : SCC refinement for language emptiness. In K. G. Larsen et M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR01)*, volume 2154 of *Lectures Notes in Computer Science*, pages 456–471. Springer-Verlag, 2001. 2.4.1

[75] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2) :72–99, 1983. 2.1.7

[76] Pierre Wolper. The tableau method for temporal logic : An overview. *Logique et Analyse*, (110–111) :119–136, 1985. 2.1.7, 2.1.7

[77] 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. 2.3.1.1

[78] Pierre Wolper et Denis Leroy. Reliable hashing without collision detection. In C. Courcoubertis, editor, *Proceedings for the 5th Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lectures Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993. 2.6.1.1

[79] 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 [72]. 2.3.1.1, 2.3.1.1

Annexe A

Les huit reines en Python et C++

Cette annexe présente la résolution du problème des huit reines avec les BDD, afin de comparer l'interface Python de BuDDy fournie par Spot avec sa cousine C++.

Il s'agit d'un problème classique de programmation par contrainte dans lequel on cherche à disposer huit reines sur un échiquier, de façon à ce que deux reines ne se trouvent jamais sur la même ligne, colonne, ou diagonale.

Le problème est ici traité dans le cas général où l'on veut placer N reines sur un plateau de $N \times N$ cases. Il est résolu en associant une variable BDD à chaque case (la variable est positive si une reine est présente, négative autrement), et en exprimant les contraintes entre toutes ces variables. Andersen [2] a un paragraphe à propos de ce problème.

A.1 C++

Cet exemple fait partie des programmes d'exemple distribués avec BuDDy.

```
#include <stdlib.h>
#include <iostream>
#include "bdd.h"
using namespace std;

int N;                /* Size of the chess board */
bdd **X;              /* BDD variable array */
bdd queen;            /* N-queen problem express as a BDD */

/* Build the requirements for all other fields than (i,j) assuming
   that (i,j) has a queen */
void build(int i, int j)
{
   bdd a=bddtrue, b=bddtrue, c=bddtrue, d=bddtrue;
   int k,l;

      /* No one in the same column */
   for (l=0 ; l<N ; l++)
      if (l != j)
         a &= X[i][j] >> !X[i][l];

      /* No one in the same row */
   for (k=0 ; k<N ; k++)
      if (k != i)
         b &= X[i][j] >> !X[k][j];

      /* No one in the same up-right diagonal */
   for (k=0 ; k<N ; k++)
   {
      int ll = k-i+j;
      if (ll>=0 && ll<N)
         if (k != i)
```

```
            c &= X[i][j] >> !X[k][ll];
   }

      /* No one in the same down-right diagonal */
   for (k=0 ; k<N ; k++)
   {
      int ll = i+j-k;
      if (ll>=0 && ll<N)
         if (k != i)
            d &= X[i][j] >> !X[k][ll];
   }

   queen &= a & b & c & d;
}


int main(int ac, char **av)
{
   int n,i,j;

   if (ac != 2)
   {
      fprintf(stderr, "USAGE:  queen N\n");
      return 1;
   }

   N = atoi(av[1]);
   if (N <= 0)
   {
      fprintf(stderr, "USAGE:  queen N\n");
      return 1;
   }

   /* Initialize with 100000 nodes, 10000 cache entries and NxN variables */
   bdd_init(N*N*256, 10000);
   bdd_setvarnum(N*N);

   queen = bddtrue;

      /* Build variable array */
   X = new bdd*[N];
   for (n=0 ; n<N ; n++)
      X[n] = new bdd[N];

   for (i=0 ; i<N ; i++)
      for (j=0 ; j<N ; j++)
         X[i][j] = bdd_ithvar(i*N+j);

      /* Place a queen in each row */
   for (i=0 ; i<N ; i++)
   {
      bdd e = bddfalse;
      for (j=0 ; j<N ; j++)
         e |= X[i][j];
      queen &= e;
   }

      /* Build requirements for each variable(field) */
   for (i=0 ; i<N ; i++)
      for (j=0 ; j<N ; j++)
       {
          cout << "Adding position " << i << "," << j << "\n" << flush;
```

```
          build(i,j);
       }

      /* Print the results */
   cout << "There are " << bdd_satcount(queen) << " solutions\n";
   cout << "one is:\n";
   bdd solution = bdd_satone(queen);
   cout << bddset << solution << endl;

   bdd_done();

   return 0;
}
```

A.2 Python

Voici le même programme écrit en Python. Il fait partie de la batterie de tests de `libspot` exécutés systématiquement par `make check`. C'est la raison pour laquelle `N` possède une valeur par défaut.

```
import sys
from buddy import *

# Build the requirements for all other fields than (i,j) assuming
# that (i,j) has a queen.
def build(i, j):
    a = b = c = d = bddtrue

    # No one in the same column.
    for l in side:
        if l != j:
            a &= X[i][j] >> -X[i][l]

    # No one in the same row.
    for k in side:
        if k != i:
            b &= X[i][j] >> -X[k][j]

    # No one in the same up-right diagonal.
    for k in side:
        ll = k - i + j
        if ll >= 0 and ll < N:
            if k != i:
                c &= X[i][j] >> -X[k][ll]

    # No one in the same down-right diagonal.
    for k in side:
        ll = i + j - k
        if ll >= 0 and ll < N:
            if k != i:
                c &= X[i][j] >> -X[k][ll]

    global queen
    queen &= a & b & c & d


# Get the number of queens from the command-line, or default to 8.
if len(sys.argv) > 1:
    N = int(argv[1])
else:
    N = 8
```

```
side = range(N)

# Initialize with 100000 nodes, 10000 cache entries and NxN variables.
bdd_init(N * N * 256, 10000)
bdd_setvarnum(N * N)

queen = bddtrue

# Build variable array.
X = [[bdd_ithvar(i*N+j) for j in side] for i in side]

# Place a queen in each row.
for i in side:
    e = bddfalse
    for j in side:
        e |= X[i][j]
    queen &= e

# Build requirements for each variable(field).
for i in side:
    for j in side:
        print "Adding position %d, %d" % (i, j)
        build(i, j)

# Print the results.
print "There are", bdd_satcount(queen), "solutions"
print "one is:"
solution = bdd_satone(queen)
bdd_printset(solution)
print

bdd_done()
```

Annexe B

Statistiques de LBTT

```
Statistics after round 100
**************************


  State space statistics
  ======================

    100 state spaces generated
    2000 states generated (20.00 states per state space)
    8447 transitions generated (84.47 transitions per state space)


  LTL formula statistics
  ======================

    100 LTL formulas generated

    Atomic symbol distribution:
         symbol  true        false       p0          p1          p2
              #  5           8           41          55          46
      #/formula  0.050       0.080       0.410       0.550       0.460

         symbol  p3          p4          p5
              #  47          47          45
      #/formula  0.470       0.470       0.450

    Operator distribution:
       operator  !           /\          <>          []          U
              #  52          40          33          45          28
      #/formula  0.520       0.400       0.330       0.450       0.280

       operator  V           X           xor         <->         ->
              #  23          43          17          18          23
      #/formula  0.230       0.430       0.170       0.180       0.230

       operator  \/
              #  45
      #/formula  0.450
```

Spot (Couvreur -- LaCIM)
========================

BÜCHI AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	1018	5826
(avg.)		10.18	58.26
Neg. formulae	100	1062	6535
(avg.)		10.62	65.35
All formulae	200	2080	12361
(avg.)		10.40	61.80

	Number of acceptance sets	Time consumed (seconds)
Pos. formulae	78	4.42
(avg.)	0.78	0.04
Neg. formulae	77	4.34
(avg.)	0.77	0.04
All formulae	155	8.76
(avg.)	0.78	0.04

PRODUCT AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	16945	108043
(avg.)		169.45	1080.43
Neg. formulae	100	18072	113820
(avg.)		180.72	1138.20
All formulae	200	35017	221863
(avg.)		175.09	1109.32

Spot (Couvreur -- LACIM), degeneralized
==========

BÜCHI AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	1500	6349
(avg.)		15.00	63.49
Neg. formulae	100	1542	7029
(avg.)		15.42	70.29
All formulae	200	3042	13378
(avg.)		15.21	66.89

	Number of acceptance sets	Time consumed (seconds)
Pos. formulae	98	3.96
(avg.)	0.98	0.04
Neg. formulae	98	3.96
(avg.)	0.98	0.04
All formulae	196	7.92
(avg.)	0.98	0.04

PRODUCT AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	26968	134061
(avg.)		269.68	1340.61
Neg. formulae	100	28478	139988
(avg.)		284.78	1399.88
All formulae	200	55446	274049
(avg.)		277.23	1370.24

Spot (Couvreur -- FM)
=====================

BÜCHI AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	1025	6135
(avg.)		10.25	61.35
Neg. formulae	100	1080	6406
(avg.)		10.80	64.06
All formulae	200	2105	12541
(avg.)		10.53	62.70

	Number of acceptance sets	Time consumed (seconds)
Pos. formulae	62	3.44
(avg.)	0.62	0.03
Neg. formulae	65	3.51
(avg.)	0.65	0.04
All formulae	127	6.95
(avg.)	0.64	0.03

PRODUCT AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	11045	60953
(avg.)		110.45	609.53
Neg. formulae	100	11813	62444
(avg.)		118.13	624.44
All formulae	200	22858	123397
(avg.)		114.29	616.99

Spot (Couvreur -- FM), degeneralized
====================================

BÜCHI AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	903	3453
(avg.)		9.03	34.53
Neg. formulae	100	868	3160
(avg.)		8.68	31.60
All formulae	200	1771	6613
(avg.)		8.86	33.06

	Number of acceptance sets	Time consumed (seconds)
Pos. formulae	98	3.30
(avg.)	0.98	0.03
Neg. formulae	98	3.38
(avg.)	0.98	0.03
All formulae	196	6.68
(avg.)	0.98	0.03

PRODUCT AUTOMATA	Number of automata	Number of states	Number of transitions
Pos. formulae	100	11623	54028
(avg.)		116.23	540.28
Neg. formulae	100	11526	51103
(avg.)		115.26	511.03
All formulae	200	23149	105131
(avg.)		115.75	525.65

```
Failures to compute Büchi automaton
===================================

  Spot (Couvreur -- LaCIM)
      Positive formulae: 0     [0.00% of 100 attempts]
      Negative formulae: 0     [0.00% of 100 attempts]
      Total:             0     [0.00% of 200 attempts]

  Spot (Couvreur -- LACIM), degeneralized
      Positive formulae: 0     [0.00% of 100 attempts]
      Negative formulae: 0     [0.00% of 100 attempts]
      Total:             0     [0.00% of 200 attempts]

  Spot (Couvreur -- FM)
      Positive formulae: 0     [0.00% of 100 attempts]
      Negative formulae: 0     [0.00% of 100 attempts]
      Total:             0     [0.00% of 200 attempts]

  Spot (Couvreur -- FM), degeneralized
      Positive formulae: 0     [0.00% of 100 attempts]
      Negative formulae: 0     [0.00% of 100 attempts]
      Total:             0     [0.00% of 200 attempts]


Model checking result consistency check failures
================================================

  Spot (Couvreur -- LaCIM)
      0     [0.00% of 100 checks performed]

  Spot (Couvreur -- LACIM), degeneralized
      0     [0.00% of 100 checks performed]

  Spot (Couvreur -- FM)
      0     [0.00% of 100 checks performed]

  Spot (Couvreur -- FM), degeneralized
      0     [0.00% of 100 checks performed]
```

Result inconsistency statistics
==========

		Spot (Couvreur -- La	Spot (Couvreur -- LA
Spot (Couvreur -- La	[1]		0/200 (0.00%)
	[2]		0/200 (0.00%)
	[3]	0/100 (0.00%)	0/200 (0.00%)
Spot (Couvreur -- LA	[1]	0/200 (0.00%)	
	[2]	0/200 (0.00%)	
	[3]	0/200 (0.00%)	0/100 (0.00%)
Spot (Couvreur -- FM	[1]	0/200 (0.00%)	0/200 (0.00%)
	[2]	0/200 (0.00%)	0/200 (0.00%)
	[3]	0/200 (0.00%)	0/200 (0.00%)
Spot (Couvreur -- FM	[1]	0/200 (0.00%)	0/200 (0.00%)
	[2]	0/200 (0.00%)	0/200 (0.00%)
	[3]	0/200 (0.00%)	0/200 (0.00%)

		Spot (Couvreur -- FM	Spot (Couvreur -- FM
Spot (Couvreur -- La	[1]	0/200 (0.00%)	0/200 (0.00%)
	[2]	0/200 (0.00%)	0/200 (0.00%)
	[3]	0/200 (0.00%)	0/200 (0.00%)
Spot (Couvreur -- LA	[1]	0/200 (0.00%)	0/200 (0.00%)
	[2]	0/200 (0.00%)	0/200 (0.00%)
	[3]	0/200 (0.00%)	0/200 (0.00%)
Spot (Couvreur -- FM	[1]		0/200 (0.00%)
	[2]		0/200 (0.00%)
	[3]	0/100 (0.00%)	0/200 (0.00%)
Spot (Couvreur -- FM	[1]	0/200 (0.00%)	
	[2]	0/200 (0.00%)	
	[3]	0/200 (0.00%)	0/100 (0.00%)

[1] Model checking result cross-comparison failures
(number of failures / number of global cross-comparisons)

[2] Model checking result cross-comparison failures (initial state only)
(number of failures / number of cross-comparisons)

[3] Büchi automata intersection emptiness check failures
(number of failures / number of checks performed)