THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Etienne RENAULT

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Contribution aux tests de vacuité pour le *model checking* explicite

Soutenue le 5 Décembre 2014 devant le jury composé de :

Laure Petrucci, Professeur à l'Université Paris XIII	Rapporteur
Francois Vernadat, Professeur à l'INSA de Toulouse	Rapporteur
Emmanuelle Encrenaz, Maître de conférences à l'Université Paris VI	Examinateur
Jean-Michel Couvreur, Professeur à l'Université d'Orléans	Examinateur
Jaco Van De Pol, Professeur à l'Université de Twente	Examinateur
Alexandre Duret-Lutz, Maître de conférences à l'EPITA	Encadrant
Denis Poitrenaud, Maître de conférences à l'Université Paris 5	Encadrant
Fabrice Kordon, Professeur à l'Université Paris VI	Directeur de thèse

Remerciements

Cadeau d'anniversaire de moi à moi ! Comment me remercier ?

Kuzco, l'empereur mégalo

Je ne peux commencer ces remerciements sans mentionner Fabrice Kordon (@fabricekordon) ~~qui me supporte~~ qui m'accompagne dans la bonne humeur depuis ma deuxième année de Licence (un temps où mon budget chocolat était bien plus élevé !). J'ai eu énormément de plaisir à travailler et à enseigner avec lui.

Je tiens aussi à remercier Alexandre Duret-Lutz pour sa disponibilité et tous ses conseils : il m'a, entre autres, appris ce qu'était véritablement que *coder* (mais ça ne veut pas forcément dire que je suis vert sur la *buildfarm* !). Nous avons partagé de nombreux mardis très agréables (je finirai bien par l'avoir un jour !).

Un grand merci à Denis Poitrenaud qui m'a encadré, abondamment conseillé, et poussé (avec force d'enthousiasme !) à aller voir *tous* les petits détails. J'apprécie énormément la façon dont il m'a appris à travailler (et aussi la façon dont il m'a appris à lâcher la pression quand il le fallait) !

Je tiens aussi à remercier mes deux rapporteurs Laure Petrucci et Francois Vernadat pour le temps qu'ils ont accordé à la relecture de ce manuscrit. De même, je suis honoré par l'intérêt qu'Emmanuelle Encrenaz, Jaco van de Pol, et Jean-Michel Couvreur ont porté à mes travaux.

Bien évidemment, je n'en serai pas là aujourd'hui sans mes différents responsables de stages qui n'ont pas hésité à me fouetter quand il le fallait (sans quoi je serai tombé dans BzFlag plutôt que dans la recherche) : Lom Hillah, Souheib Baarir, Alban Linard et Alexandre Hamez.

Je remercie Maximilien Collange (même si je ne sais toujours pas s'il est convaincu ou non que $P \neq NP$) pour sa bonne humeur et ses questions sur tout (et rien !).

S'il n'en reste qu'une au Bureau 818, Laure Millet serait celle là ! Elle est ma plus fidèle cobureau et je ne peux concevoir ce que vont devenir les lundis et les vendredis sans nos causettes. (Je n'ose imaginer le temps que je lui ai fait perdre.).

Je veux aussi remercier Edwin Carlinet, mon thésard dual venu des images, pour son soutien (et pas seulement quand il s'agit d'aller au Kebab !) ainsi que sa bonne humeur et ses blagues (même si je pense que j'en ai plus à mon actif !). Je ne doute pas qu'un jour il trouvera son chemin dans les graphes.

En parlant de graphes et d'automates, je me dois de mentionner Ala Eddine Ben-Salem, mon thésard jumeau, qui est un grand bavard (malgré les apparences) et avec qui j'ai probablement le plus échangé durant ces trois années (et ce n'est pas parce qu'il est *stuttering* !).

Les choix que l'on fait mènent parfois aux mêmes endroits. C'est comme ça que j'ai pu retrouver avec plaisir Clément Démoulins (aka Ensu) au lrde ce qui nous a permis de recréer des liens (de niveau 6 et plus !).

Je veux aussi remercier dans le désordre les gens du lrde et du lip6 (je suis certain que j'en oublie, pardonnez moi !) qui m'ont accompagné au fil des années : Olivier Ricou (sans qui cette thèse n'aurait pas été possible), Cédric Besse (pour son humour), Jean-Luc Mounier (pour ses nombreux sauvetages de machines et ses discussions sur MacOSX), Nicolas Gibelin (enfin parti pour ses montagnes), Mathieu Sassolas (pour son tikz et Jean-Jacques), Akim Demaille (toujours prêt à discuter technique), Yann Thierry mieg (toujours à la recherche d'un café), Béatrice Bérard (pour sa bonne humeur), Thierry Géraud (maître incontesté des jeux de mots), Daniela Beker (à ma rescousse à chaque fois que les problèmes administratifs me submergeaient), Myriam ROBERT-SEIDOWSKY (sans qui on ne saurait pas qui vient chanter en bas !), Yoann LAURENT (pour ses discussions de bout de couloir), Jonathan FABRIZIO (à qui j'ai « volé » le cluster bien trop de fois) . . . et tous les autres !

Je veux plus particulièrement remercier Harris BAKIRAS© (qui est parti vers d'autres destinations) et Florian DAVID (avec qui j'ai partagé *bien plus* – graouh – que la rue Saint Anne) qui me font passer de très (très) bons moments depuis 5 ans !

Une pensée aussi à tous les ovillois (qui ne le sont plus depuis) et qui se reconnaîtront sans problème (scout toujours mais pas moi) : Seb, Nico, Clément, Simon, . . .

Enfin, un ÉNORMISSIME merci à toute ma famille (mes parents, mon frère Xavier, ma sœur Élise, ma belle soeur Claire) pour leur soutien sans faille et pour tout ce qu'ils m'apportent depuis toutes ces années (et pas seulement en nourriture !). Ils m'ont permis d'aller serein (et sans les petites roues) jusqu'à ce dernier jour d'école !

Toutes mes pensées vont finalement vers Marine (la plus belle des rencontres) et qui a été d'un soutien absolu pendant ces trois années (et qui fait mon bonheur jour après jour).

Table des matières

Introduction générale

A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.

Alan M. Turing

Sommaire

Cette introduction détaille les enjeux de la vérification *de systèmes discrets et positionne cette thèse par rapport aux approches existantes. L'objectif est d'introduire notre cadre de travail qui sera plus largement détaillé dans les chapitres 1 et 2.*

Contexte et problématique

Ces dernières années, le développement massif des technologies de communication a banalisé l'utilisation de systèmes réactifs, concurrents ou répartis. Les systèmes réactifs répondent constamment aux sollicitations de leur environnement et considèrent la propagation de l'information comme instantanée. Les systèmes concurrents sont composés de plusieurs tâches s'exécutant en parallèle. Les systèmes répartis, quant à eux, se focalisent sur l'interaction de différents processus sur des sites distincts.

Tous ces systèmes sont particulièrement utilisés dans des domaines critiques (médical, avionique, transport, etc.) pour lesquels une défaillance peut avoir des conséquences colossales en termes humains et financiers.

L'élaboration de systèmes sûrs constitue alors un véritable enjeu qui doit être pris en compte dès la phase de conception. Plusieurs approches existent pour *vérifier* si un système satisfait un ensemble de *propriétés* :

Les tests **:** il s'agit probablement de la méthode la plus facile à mettre en œuvre et la plus employée. Une première étape consiste à élaborer des scénarios d'exécution dont les résultats sont connus à l'avance. Le système est ensuite exécuté en respectant ces scénarios et l'on peut vérifier s'il réagit de la façon attendue. La présence d'un tel jeu de tests est rassurante mais ne garantit en rien son exhaustivité, i.e. la validité des comportements en dehors des cas testés. Certains travaux s'intéressent néanmoins à maximiser la couverture de ces tests.

La vérification par preuves **:** dans cette approche, le système est exprimé sous la forme de formules de logique. Celles-ci sont ensuite combinées pour construire de nouveaux lemmes jusqu'à déduire ou invalider la propriété à vérifier. D'autre part, la mise en place des axiomes sur lesquels la dérivation est basée n'est généralement pas automatique : un utilisateur expert doit interagir pour orienter correctement la preuve.

Le model checking **:** cette approche est entièrement automatique dès lors que l'on possède un ensemble de propriétés et une abstraction équivalente au système (aussi connue sous le nom de modèle, cf. chapitre 1). L'idée est de parcourir cette abstraction pour vérifier si les propriétés y sont valides. Un des avantages de cette technique est sa capacité à extraire des comportements invalidants les propriétés. Ainsi elle peut être utilisée dès les phases de conception pour valider les prototypes. Un autre avantage est de pouvoir s'appliquer sur les sources du système à vérifier (sous certaines hypothèses) : cela permet une vérification en continu pendant la phase de développement.

Ces techniques ont chacune leurs forces et leurs faiblesses : les tests ne sont pas exhaustifs et ne peuvent pas être appliqués dès la phase de conception mais sont faciles à mettre en œuvre ; l'utilisation de preuves est difficile à mettre en place puisqu'elle nécessite la présence d'un expert mais est exhaustive, applicable dès la phase de conception, et gère efficacement les systèmes paramétrés ; enfin, le *model checking* est exhaustif et peut être automatique bien que la vérification de systèmes paramétrés soit moins efficace que par l'utilisation de preuves, puisque généralement limitée à des sous-instances de modèles.

La vérification du « bon » fonctionnement d'un système requiert toujours une intervention humaine à un moment donné : l'écriture des tests, la mise en place des axiomes, la spécification des propriétés à vérifier. La diffusion des méthodes de vérification nécessite que cette intervention soit la moins coûteuse possible et que la vérification soit exhaustive. La mise en place de normes avioniques recommandant l'utilisation du *model checking* (norme DO178C) et l'attribution du prix Turing à Pnueli en 1996, puis à Clarke, Emerson et Sifakis en 2007 pour leur travaux sur le *model checking* témoignent que cette approche est suffisamment reconnue pour être utilisée dans un contexte industriel.

Le *model checking* souffre néanmoins de deux problèmes majeurs : la mémoire requise par les algorithmes de vérification et leur temps d'exécution. En effet, l'exploration exhaustive requiert l'analyse de l'ensemble des comportements possibles, ce qui est à la fois long et fortement coûteux en mémoire, même pour de « petits » systèmes. Afin que ces méthodes soient intégrées dans des contextes industriels, il est nécessaire de réduire cette durée d'exécution pour avoir des cycles de conception rapides. Ces problèmes ont été abordés selon deux grands axes :

Les techniques explicites **:** ces techniques explorent et stockent l'intégralité des états du système qui est représenté explicitement sous la forme d'un automate. De nombreuses optimisations, dédiées ou non aux systèmes concurrents, existent pour améliorer ces techniques (cf. chapitre 2).

Les techniques symboliques **:** ces techniques utilisent une représentation compacte du système basée sur des ensembles. Ces techniques sont souvent plus efficaces pour les systèmes concurrents, distribués ou répartis, car elles font intervenir des mécanismes généralement trop coûteux pour l'analyse de systèmes séquentiels.

Les techniques symboliques ont une efficacité qui dépend donc du système et nous ne les considérerons pas ici[1]. Dans cette thèse, nous nous focalisons sur les techniques explicites et notamment sur les moyens disponibles pour accélérer le processus de vérification. Plus généralement nous tentons de savoir quelles sont les techniques les plus efficaces pour vérifier un système dans le cadre des techniques explicites pour le *model checking*. Nous porterons une attention particulière au support de l'équité qui reste relativement peu étudié dans la littérature.

Contributions

Les contributions de ce travail portent sur l'amélioration des algorithmes de vérification pour le *model checking*. Comme ils peuvent être réalisés aussi bien de manière séquentielle que de manière parallèle, ce manuscrit est composé de deux grandes parties les traitant.

Amélioration des algorithmes séquentiels. Cette première partie étudie les algorithmes de vérification et montre comment ils peuvent être optimisés. Une nouvelle approche basée sur l'utilisation d'une structure d'*union-find* y est aussi présentée. Cette structure permet de partitionner efficacement des ensembles et s'intègre parfaitement aux algorithmes de vérification tout en permettant de réduire la complexité dans le pire cas.

Amélioration des algorithmes parallèles. Cette seconde partie montre tout d'abord que l'automate de la propriété peut être décomposé pour accélérer la procédure de vérification. Ensuite, un aperçu des algorithmes existants est dressé et certaines métriques sont proposées pour les comparer, tant sur le passage à l'échelle que sur la redondance. L'idée d'utiliser une structure d'union-find est ensuite réutilisée pour favoriser le partage d'informations et accélérer la vérification. Cette approche inédite permet de combiner plusieurs algorithmes.

Plan du mémoire

Ce mémoire est découpé en trois grandes parties.

Partie I : Préliminaires Le chapitre 1 présente les définitions nécessaires à la compréhension de ce manuscrit tandis que le chapitre 2 introduit le *model checking explicite*. Ces deux chapitres servent de base à l'ensemble du manuscrit et justifient l'utilisation d'automates généralisés.

Partie II : Contributions aux tests de vacuité séquentiels Le chapitre 3 détaille les principaux algorithmes de vérification et montre comment la propriété à vérifier peut impacter leurs performances. Le chapitre 4 montre leurs limitations et les algorithmes qui permettent de les contourner. Ces derniers intègrent pour la première fois une optimisation présentée à l'origine sur les graphes ainsi qu'une optimisation permettant d'économiser de la mémoire. Le chapitre 5 introduit la structure d'union-find et présente deux nouveaux algorithmes qui en tirent parti. Enfin, le chapitre 6 compare les performances de tous ces algorithmes.

1. Les techniques présentées au chapitre 7 y sont néanmoins directement applicables puisque basées sur l'analyse de la structure de l'automate de la formule à vérifier.

Partie III : Contributions aux tests de vacuité parallèles Le chapitre 7 montre que ces algorithmes de vérification peuvent exploiter l'automate de la formule en le décomposant en plusieurs automates. Le chapitre 8 présente les algorithmes de vérification parallèles et introduit les métriques permettant de les comparer : des limitations apparaissent alors sur ces algorithmes. Le chapitre 9 montre comment elles peuvent être dépassées via l'utilisation d'une structure d'union-find. Ces différentes approches sont ensuite comparées dans le chapitre 10.

Enfin, le dernier chapitre présente les perspectives ouvertes par cette thèse. Plusieurs axes de recherche y sont évoqués et une combinaison de tous les travaux de cette thèse y est présentée.

Première partie

Préliminaires

Chapitre 1

La représentation de l'espace d'état d'un système et d'une propriété

All models are wrong, but some are useful.

George E. P. Box

Sommaire

Ce chapitre a pour objectif l'introduction des définitions et des notations nécessaires à la compréhension de ce manuscrit. Ce travail s'inscrit dans le cadre de la vérification par model checking *au moyen d'automates de Büchi généralisés basés sur les transitions. Ce chapitre motive ce choix et montre comment ils peuvent être utilisés pour représenter aussi bien les états du système que les propriétés que ce dernier doit vérifier.*

1.1 Le modèle

Un système est un ensemble de composants qui collaborent à la réalisation d'une tâche. Chaque composant est constitué de variables dont l'évolution impacte le comportement du système. À tout instant on peut capturer l'*état* d'un système, i.e. la valeur de chacune des variables. L'*espace d'état* se définit comme l'ensemble des états possibles.

L'augmentation du nombre de variables conduit à une explosion du nombre d'états : ce phénomène s'appelle l'*explosion combinatoire*. Par la suite, nous ne considérons que les systèmes ayant un nombre fini de variables prenant des valeurs dans des domaines eux-mêmes finis : cela permet de ne manipuler que des systèmes ayant un nombre fini d'états. Cette restriction est courante, notamment pour les systèmes embarqués qui ont de fortes contraintes mémoire.

La *modélisation* a comme objectif la création d'un *modèle* abstrait du système. Ce dernier doit être plus simple (moins d'états) mais doit avoir le même comportement vis-à-vis des propriétés que l'on souhaite étudier. L'analyse de ce modèle contribue alors à combattre l'explosion combinatoire. Cette section décrit les formalismes permettant de représenter aussi bien ce modèle que les propriétés à vérifier dessus.

Prenons comme « fil rouge » de cette section le problème de la terminaison d'un système multi-processus au travers d'un exemple simple : la classe d'Archimède. Informellement ce problème peut être décrit de la manière suivante :

Exemple.
Archimède pose un problème à une classe d'étudiants qui peuvent soit y réfléchir soit lire ce qui est écrit au tableau. Tout étudiant peut partager une idée avec la classe en allant l'écrire au tableau qui est assez grand pour que tous les étudiants puisse y écrire en même temps. Dès qu'un étudiant trouve la solution il crie « eurêka ! » et la classe se termine immédiatement. Enfin un étudiant peut juger le problème trop dur et abandonner : il doit alors attendre que la solution soit trouvée ou que tous les étudiants abandonnent pour que la classe se termine.

Quatre états distincts peuvent être définis pour modéliser un étudiant :

ReadingOrThinking	: l'étudiant réfléchit au problème et peut lire ce qui est au tableau ;
Writing	: l'étudiant écrit au tableau ;
Waiting	: l'étudiant a abandonné et attend la fin du cours ;
Done	: la classe est terminée, l'étudiant peut en sortir.

Ces états sont caractérisés par des variables spécifiques appelées *propositions atomiques*.

1.1.1 Les propositions atomiques

Une *proposition* (ou *variable propositionnelle*) permet de capturer l'état d'une entité à un instant donné et peut prendre deux valeurs : « vrai » ou « faux ». Ces variables peuvent être combinées entre-elles pour former de nouvelles propositions. Lorsqu'une proposition n'est pas issue d'une telle combinaison, il s'agit d'une *proposition atomique*, par exemple : « l'étudiant travaille ». L'ensemble non vide et fini des variables propositionnelles atomiques d'un système est traditionnellement noté AP (*Atomic Propositions*). La notation 2^{AP} représente l'ensemble des parties de AP. Par exemple, si $AP = \{a, b\}$, alors $2^{AP} = \{\, \emptyset,\ \{a\}, \{b\}, \{a,\ b\}\}$.

Un *littéral* est une proposition atomique ou sa négation. Ainsi $a, \neg\, a, b$, et $\neg\, b$ sont les littéraux de $AP = \{a,\ b\}$. Un *cube* est une conjonction de littéraux ; si toutes les propositions atomiques de AP apparaissent dans un cube, ce dernier est un *minterm*. Si $|AP|$ représente la *cardinalité* de AP, alors il existe $2^{|AP|}$ minterms basés sur cet ensemble. Comme il existe une bijection entre l'ensemble des minterms et l'ensemble des parties de AP, la notation 2^{AP} fait référence indifféremment à l'un ou l'autre de ces deux ensembles. Par exemple, sur $AP = \{\mathrm{a, b}\}$, il existe la bijection suivante :

$$\begin{aligned} \{a, b\} &\iff a \wedge b & \qquad \{a\} &\iff a \wedge \neg b \\ \{b\} &\iff \neg a \wedge b & \emptyset &\iff \neg a \wedge \neg b \end{aligned}$$

Les *formules propositionnelles* peuvent être vues comme des disjonctions de minterms. L'*ensemble des formules propositionnelles* est noté $2^{2^{AP}}$ car il existe une injection entre l'ensemble des parties de 2^{AP} et l'ensemble des formules propositionnelles. La façon la plus simple de voir cette injection est d'écrire la table de vérité de l'ensemble 2^{AP} et de faire une disjonction entre les éléments à 1 pour une ligne donnée. Par exemple si $AP = \{a,\, b\}$, alors la formule propositionnelle $(\neg\, a \wedge \neg\, b) \vee (\neg\, a \wedge b) \equiv \neg\, a$ peut être exprimée sous la forme $\{\, \emptyset,\, \{\, b\,\}\}$.

De façon plus générale, les formules propositionnelles sont définies inductivement sur un ensemble AP non vide tel que :

1. toute proposition atomique $a \in AP$ est une formule propositionnelle ;
2. si φ est une formule propositionnelle alors la *négation* $\neg\,\varphi$ l'est aussi ;
3. si φ_1 et φ_2 sont des formules propositionnelles alors la *disjonction* $\varphi_1 \vee \varphi_2$ l'est aussi ;
4. si φ_1 et φ_2 sont des formules propositionnelles alors la *conjonction* $\varphi_1 \wedge \varphi_2$ l'est aussi ;
5. si φ_1 et φ_2 sont des formules propositionnelles alors le *ou-exclusif* $\varphi_1 \oplus \varphi_2$ l'est aussi ;
6. si φ_1 et φ_2 sont des formules propositionnelles alors l'*implication* $\varphi_1 \rightarrow \varphi_2$ l'est aussi ;
7. si φ_1 et φ_2 sont des formules propositionnelles alors l'*équivalence* $\varphi_1 \leftrightarrow \varphi_2$ l'est aussi.

Cette définition autorise des raccourcis d'écriture tels que $\neg\, a$ pour exprimer la disjonction de minterms : $(\neg\, a \wedge b) \vee (\neg\, a \wedge \neg\, b)$ sur $AP = \{a, b\}$. La variable b est alors dite *muette*. La constante $\top$ signifie que toutes les variables sont muettes et peut s'interpréter comme la disjonction de tous les minterms de 2^{AP} (respectivement $\perp$ peut s'interpréter comme la conjonction de deux minterms différents). Notons aussi que les points 4 à 7 ne sont qu'une facilité d'écriture puisque :

$$\begin{aligned} \varphi_1 \wedge \varphi_2 &\overset{def}{=} \neg\,(\neg\,\varphi_1 \vee \neg\,\varphi_2) \\ \varphi_1 \rightarrow \varphi_2 &\overset{def}{=} \neg\,\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 &\overset{def}{=} (\neg\,\varphi_1 \wedge \neg\,\varphi_2) \vee (\varphi_1 \wedge \varphi_2) \\ \varphi_1 \oplus \varphi_2 &\overset{def}{=} (\neg\,\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \neg\,\varphi_2) \end{aligned}$$

Chaque variable du modèle peut être convertie en un ensemble de propositions atomiques. Par exemple, une variable entière peut simplement être représentée par un ensemble de propositions atomiques représentant chaque bit. À tout instant, un état du modèle est étiqueté par un minterm et deux états différents peuvent être représentés par le même minterm.

Exemple.
Pour modéliser un étudiant de la classe d'Archimède quatre propositions atomiques peuvent être utilisées, chacune représentant un état. Ainsi, les propositions *working*, *writing*, *waiting* et *idle* permettent respectivement de représenter les états ReadingOrThinking, Writing, Waiting ou Done. Dans cet exemple, quand l'étudiant écrit (*writing*) il est dans l'état *Writing*. L'utilisation de quatre propositions atomiques pour cette modélisation a été faite dans un soucis de clarté mais seules deux variables auraient pu être utilisées pour distinguer ces états.

1.1.2 Les langages

L'ensemble 2^{AP} des minterms peut être vu comme les *lettres* (ou *symboles*) d'un ensemble fini appelé *alphabet* et noté Σ.

Si $\mathbb{N}$ représente l'ensemble des entiers, et $\omega \notin \mathbb{N}$ le premier ordinal infini, une *séquence* est une fonction $\sigma : \mathbb{N} \cup \{\omega\} \rightarrow \Sigma$ qui associe à un indice un élément de Σ. Un *mot* sur Σ est une séquence finie (ou non) de symboles de Σ et peut être noté $\rho = \alpha_0\alpha_1\alpha_2\ldots$ avec $\alpha_i \in \Sigma$. La notation ρ_i référence la i^{e} position de la séquence, la notation $\rho_{i\ldots}$ (resp. $\rho_{\ldots i}$) référence la séquence suffixe (resp. préfixe) de ρ à partir de (resp. jusqu'à) la lettre à la i^{e} position. La *taille* d'un mot caractérise le nombre de symboles qui le compose. Le mot de taille 0 est noté ε et est appelé *mot vide* ; un mot infini est noté ω*-mot* et a une taille ω.

Par la suite, Σ^* représente l'ensemble des mots finis sur Σ, Σ^ω l'ensemble des ω-mots, et $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ l'ensemble des mots finis non-vides. Un *langage fini* $\mathscr{L} \subseteq \Sigma^*$ est un ensemble fini de mots sur Σ ; un ω-langage $\mathscr{L}_1$ est un ensemble fini de mots infinis, i.e., $\mathscr{L}_1 \subseteq \Sigma^\omega$. Comme ce sont les ω-langages nous intéressent, nous utiliserons le terme de langage pour parler d'ω-langage, et nous préciserons quand il s'agit de langages de mots finis.

Les langages étant des ensembles, toutes les opérations ensemblistes leur sont applicables. On peut donc définir les opérations d'*union*, d'*intersection* et de *complémentation*. Soient $\mathscr{L}_1$ et $\mathscr{L}_2$ deux langages sur Σ^ω, on peut donc définir :

$$\begin{aligned}
\text{Union :} \quad & \mathscr{L}_1 \cup \mathscr{L}_2 = \{u \in \Sigma^\omega \mid u \in \mathscr{L}_1 \text{ ou } u \in \mathscr{L}_2\} \\
\text{Intersection :} \quad & \mathscr{L}_1 \cap \mathscr{L}_2 = \{u \in \Sigma^\omega \mid u \in \mathscr{L}_1 \text{ et } u \in \mathscr{L}_2\} \\
\text{Complémentation :} \quad & \overline{\mathscr{L}_1} = \{u \in \Sigma^\omega \mid u \notin \mathscr{L}_1\}
\end{aligned}$$

Les langages construits sur 2^{AP} sont particulièrement intéressants ici car ils capturent la suite des états représentants les comportements du modèle. La représentation d'un langage peut être faite au travers de divers formalismes avec des pouvoirs d'expression différents. Dans l'approche automate pour le *model checking* explicite, les structures de Kripke (cf. section 1.1.3) et les automates de Büchi (cf. section 1.2.2) sont les formalismes les plus utilisés.

1.1.3 Systèmes de transitions et structures de Kripke

Les automates de Büchi et les structures de Kripke sont basés sur un format de représentation compact : les systèmes étiqueté sur les transitions (LTS). Ces structures sont constituées d'un ensemble de sommets appelés *états* reliés par des arcs appelés *transitions*. Chaque transition est annotée par une *action*, et un état est distingué en tant qu'*état initial*. Cette restriction à un unique état initial peut facilement être contournée en créant un état initial artificiel qui va relier tous les états initiaux.

Définition 1 – Labelled Transition System (LTS)

Un *système étiqueté sur les transitions* $G = \langle\, Q,\, q_0, Act,\, \Delta \,\rangle$ est défini par :

Q	:	un ensemble fini d'états ;
$q_0 \in Q$	:	l'état initial ;
Act	:	un ensemble d'actions ;
$\Delta \subseteq Q \times Act \times Q$	:	la relation de transition donnée sous la forme de triplets (état origine, action, état destination) appelés *transitions*. Si $(s, \alpha, d) \in \Delta$, on note $s \xrightarrow{\alpha} d$.

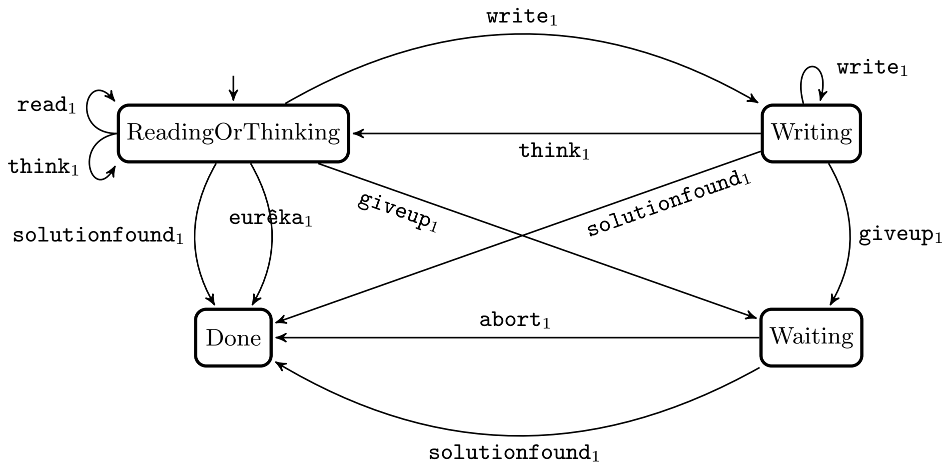


Figure 1.1 – LTS représentant un étudiant de la classe d'Archimède avec $Act = \{$ `write`$_1$, `think`$_1$, `read`$_1$, `solutionfound`$_1$, `giveup`$_1$ $\}$.

Exemple.
La figure 1.1 présente un LTS capturant le comportement d'un étudiant de la classe d'Archimède. On y retrouve les quatre états définis page 16, à savoir : ReadingOrThinking, Writing, Waiting et Done. Par convention, l'*état initial* ReadingOrThinking est indiqué par une flèche entrante sans source. Les actions possibles de ce modèle sont `read`$_1$, `write`$_1$, `think`$_1$, `eurêka`$_1$, `giveup`$_1$, `abort`$_1$ et `solutionfound`$_1$ qui représentent respectivement les actions de lire, d'écrire, de penser, de trouver la solution, d'abandonner, ou de terminer la classe. Les actions `eurêka`$_1$, `giveup`$_1$ et `abort`$_1$ ont uniquement une importance lorsque l'on souhaite modéliser et synchroniser plusieurs étudiants.

Dans un LTS les actions sont présentes à titre informatif et ne sont donc pas nécessaires à la modélisation des comportements du système. Elles sont néanmoins importantes lorsque l'on souhaite définir des règles de synchronisation pour combiner plusieurs LTS. Dans ce cas, ces étiquettes indiquent quelles actions doivent avoir lieu simultanément.

Ces règles de synchronisation sont généralement exprimées au sein d'une *table de synchronisation* qui spécifie quelles sont les actions qui doivent se synchroniser. Toutes les autres actions peuvent évoluer indépendamment : à chaque instant un processus peut progresser pour passer d'un état à un autre.

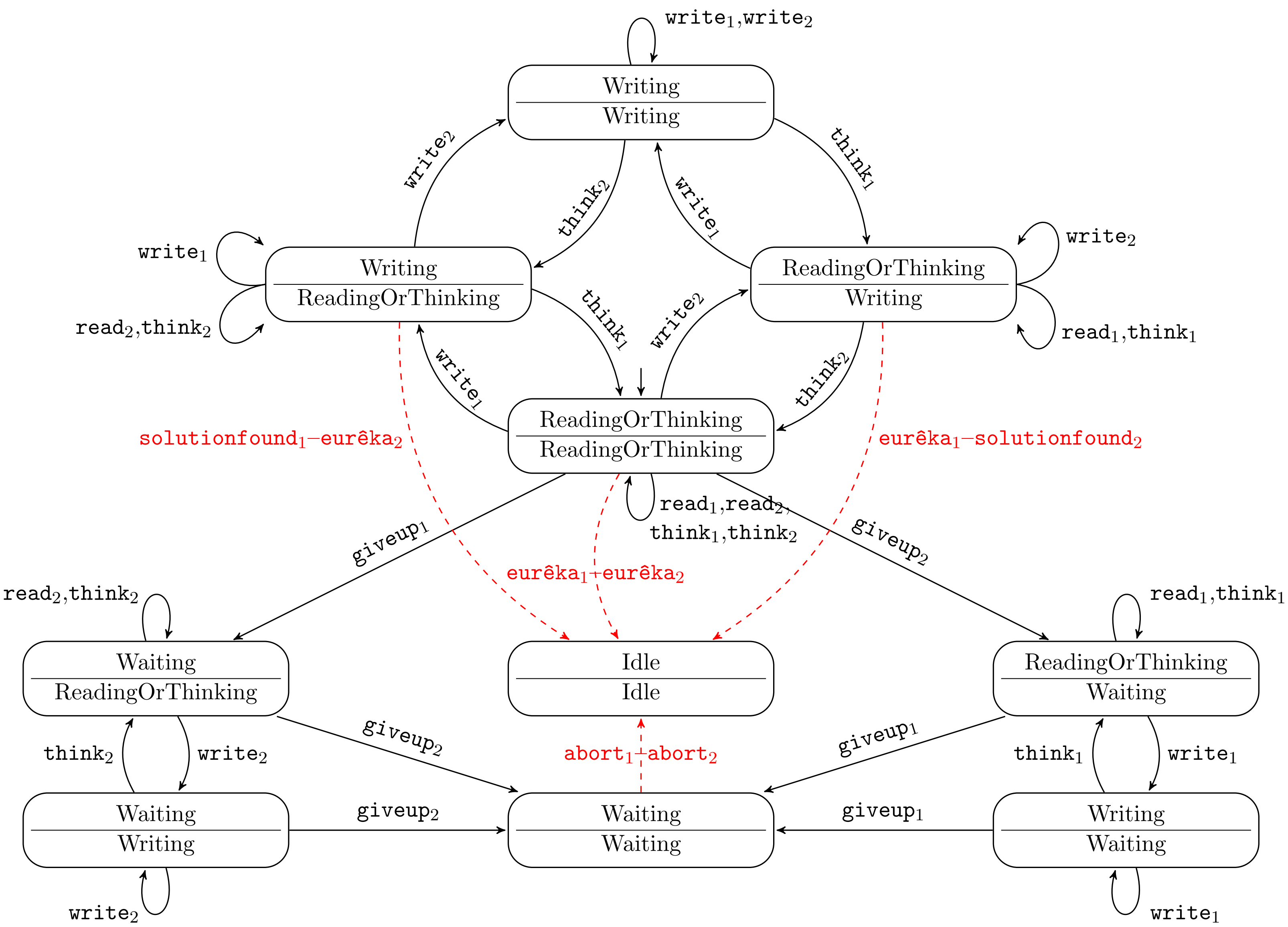


Figure 1.2 – lts pour un système composé de deux étudiants.

Exemple.

Étudiant$_1$	Étudiant$_2$
`abort`$_1$	`abort`$_2$
`eurêka`$_1$	`eurêka`$_2$
`eurêka`$_1$	`solutionfound`$_2$
`solutionfound`$_1$	`eurêka`$_2$

La table ci-dessus présente les règles de synchronisation d'un système composé de deux étudiants. Ainsi la transition `abort` ne peut être franchie que si tous les étudiants la franchissent simultanément. Lorsqu'un étudiant trouve la solution, deux possibilités existent : soit l'autre étudiant a lui aussi trouvé la solution et les deux étudiants franchissent simultanément la transition `eurêka`, soit l'autre étudiant n'a pas trouvé la solution et il emprunte la transition `solutionfound` dès que la transition `eurêka` est franchie par l'autre étudiant.

Le LTS de la figure 1.2 montre le système résultant d'une telle synchronisation. Pour plus de lisibilité, les actions portées sur les transitions sont post-fixées par 1 ou 2 pour indiquer s'il s'agit d'une action affectant le premier ou le second étudiant. Les transitions en pointillés rouges représentent les synchronisations entre les deux étudiants, elles sont annotées par leurs actions respectives. Enfin, chaque état est divisé en deux parties, la partie haute représentant l'état dans lequel se situe le premier étudiant, la partie basse indiquant l'état du second étudiant.

Définition 2 – Structure de Kripke

Une *structure de Kripke* $\mathcal{K} = \langle\, Q,\ q_0,\ Act,\ \Delta,\ AP,\ L \,\rangle$ est définie par :

$\langle\, Q,\ q_0,\ Act,\ \Delta \,\rangle$	:	un LTS tel que $Act = \emptyset$;
AP	:	un ensemble de propositions atomiques ;
$L : Q \to 2^{AP}$	:	une fonction d'interprétation associant un minterm à chaque état, i.e. l'ensemble des propositions atomiques qui y sont satisfaites.

Comme $Act = \emptyset$, si $(s, \alpha, d) \in \Delta$ on a $\alpha = \emptyset$: cette transition est alors notée $s \to d$.

Les structures de Kripke permettent de représenter l'espace d'état induit par un modèle. Dans cette structure, chaque état est étiqueté par un minterm. Les transitions indiquent alors les changements d'états et symbolisent l'évolution du modèle.

Notons ici la proximité entre la représentation du modèle (sous la forme d'un LTS) et l'espace d'état qu'il induit (sous la forme d'une structure de Kripke) : le LTS donne directement les états et les transitions qui vont constituer la structure de Kripke. Il existe cependant d'autres formalismes pour modéliser un système [74] : Promela, réseaux de Petri, ... Dans ce cas, l'analyse du modèle permet de générer l'espace d'état sous la forme d'une structure de Kripke. Dans ce manuscrit, une structure de Kripke ne diffère donc d'un LTS que par l'étiquetage des états par un minterm et par son absence d'actions sur les transitions.

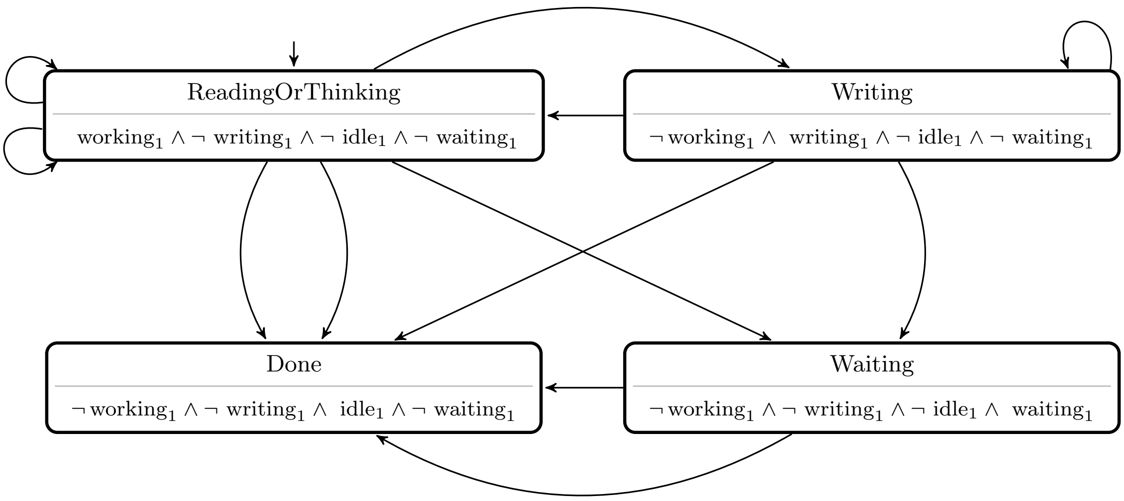


Figure 1.3 – Abstraction d'un étudiant de la classe d'Archimède avec $AP = \{\text{working}_1, \text{writing}_1, \text{idle}_1, \text{waiting}_1\}$.

Exemple.
La figure 1.3 présente la structure de Kripke pour une classe composée d'un unique étudiant. Cette structure est à rapprocher du LTS de la figure 1.1 et des propositions atomiques proposées page 18. Chaque état est divisé en deux parties : la partie haute donne le nom de l'état, la partie basse représente la valuation des propositions atomiques de cet état par la fonction L. Ainsi, $L(\text{Waiting}) = \neg\,\text{working}_1 \wedge \neg\,\text{writing}_1 \wedge \neg\,\text{idle}_1 \wedge \text{waiting}_1$.

Définition 3 – Exécution et langage d'une structure de Kripke

Soit une structure de Kripke $\mathcal{K} = \langle Q, q_0, Act, \Delta, AP, L\rangle$. Une *exécution* de $\mathcal{K}$ est une séquence infinie de minterms $w_1, \ldots, w_i, \ldots$ telle qu'il existe une suite infinie de transitions $\pi = s_1 \to d_1 \to \ldots \to s_i \to d_i \to \ldots$ avec $s_1 = q_0$, $d_i = s_{i+1}$ et $L(d_i) = w_i$.

Toute exécution est un ω-mot sur $(2^{AP})^\omega$ et le langage $\mathscr{L}_{\mathcal{K}}$ d'une structure de Kripke correspond à l'ensemble des exécutions possibles de $\mathcal{K}$. Le langage $\mathscr{L}_{\mathcal{K}}$ est *non-vide* s'il existe un circuit accessible depuis q_0, sinon il est *vide*.

Exemple.
Considérons une classe composée d'un unique étudiant dont la structure de Kripke est présentée figure 1.3. La séquence π bouclant sur Writing telle que $\pi = \text{ReadingOrThinking} \to \text{Writing} \to \text{Writing} \to \ldots$ est une exécution. Notons aussi que le minterm $\text{working} \wedge \text{writing} \wedge \text{idle} \wedge \text{waiting}$ n'est pas présent dans cette structure de Kripke puisque c'est un état non supporté : il s'agit donc d'un état non-accessible (cf. définition 5).

De manière plus générale, un LTS (ou une structure de Kripke) est un graphe orienté enraciné dans lequel chaque arc est annoté par une action : lorsque ces actions sont ignorées la théorie des graphes peut donc y être appliquée.

Définition 4 – Chemins et cycles

Pour un LTS $G = \langle\, Q,\, q_0,\, Act,\, \Delta \,\rangle$, un *chemin* de taille $n \geq 1$ entre deux états $q \in Q$ et $q' \in Q$ est une séquence finie de transitions $\rho = (s_1, \alpha_1, d_1) \ldots (s_n, \alpha_n, d_n) \;\in \Delta^+$ telle que $s_1 = q$, $s_n = q'$, et $\forall i \in \{\, 1, \ldots, n-1 \}$, $d_i = s_{i+1}$.

L'existence d'un tel chemin est notée $q \rightsquigarrow q'$. Lorsque $q = q'$ le chemin est appelé *cycle* ou *circuit*. Un cycle de taille 1 est appelé *boucle* : il s'agit simplement d'un état avec une transition qui revient sur lui même.

Définition 5 – États accessibles

Soit $G = \langle\, Q,\, q_0,\, Act,\, \Delta \,\rangle$ un LTS. Toute transition $(q,\, \alpha,\, q') \in \Delta$ implique l'existence d'un chemin entre q et q'. Tout état q est dit *accessible* s'il existe un chemin entre l'état initial et q (*i.e.* $q_0 \rightsquigarrow q$). Respectivement, un état est dit *non-accessible* s'il n'existe pas de chemin menant de l'état initial à cet état.

1.2 L'expression des propriétés

Savoir si un état est accessible ne suffit pas à assurer qu'un système est correct ; on souhaiterait pouvoir tester des comportements tels que : « Un étudiant qui travaille finit toujours par attendre » pour l'exemple donné figure 1.3. Ces comportements peuvent être exprimés au travers d'une propriété qui représente un ensemble d'exécutions satisfaisantes : toutes les exécutions du modèle doivent alors appartenir à cet ensemble. En 1977, Lamport [56] distingue deux grandes catégories de propriétés :

- *Sûreté* : exprime qu'une propriété doit être vérifiée pour toutes les exécutions du système. Une propriété de sûreté permet de s'assurer qu'un « évènement mauvais » ne se produira jamais. (Une évènement est dit mauvais si l'on ne souhaite pas qu'il survienne dans le système.)
- *Vivacité* : exprime qu'une propriété finira par être vérifiée pour toutes les exécutions du système à partir d'un certain point. Une propriété de vivacité permet de s'assurer qu'un « bon évènement » finira par arriver. (Un évènement est dit bon si on souhaite qu'il survienne dans le système.)

Plus précisément, une propriété de vivacité n'interdit aucun préfixe d'exécution tandis qu'une propriété de sûreté interdit certains préfixes et permet l'expression d'invariants sur le système.

Pour exprimer ces deux types propriétés des *logiques modales* ont été définies. Celles-ci sont basées sur la logique propositionnelle à laquelle ont été rajoutés des *opérateurs temporels* pour pouvoir considérer aussi bien les comportements infinis tels que « infiniment quelque chose survient » que les comportements finis tels que « la i$^{\text{e}}$ chose qui survient doit être … ». Ces opérateurs déterminent le modèle de temps considéré. Nous nous intéressons ici aux *logiques temporelles linéaires* qui considèrent les exécutions du système dans lesquelles le « futur » d'un état est représenté comme une séquence infinie. Ces logiques sont particulièrement adaptées à notre problématique puisqu'elles permettent l'expression de propriété d'équité (détails section 2.2) et offrent la possibilité d'extraire des exécutions invalidant la propriété. Dans un cadre pratique nous utiliserons LTL (*Linear-time Temporal Logic*) [69] qui passe pour être la plus intuitive.

Dans ce manuscrit, nous utilisons LTL comme cadre de travail mais tous les travaux présentés ici couvrent la logique PSL qui est plus expressive mais moins intuitive [1].

1.2.1 Logique temporelle à temps linéaire

La logique LTL a été introduite en 1977 par Amir Pnueli [69] pour spécifier l'évolution des systèmes en fonction du temps. Basée sur de la logique propositionnelle, la logique LTL est définie sur un ensemble de propositions atomiques AP, et la syntaxe d'une formule φ est décrite formellement de la manière suivante :

$$\varphi ::= \top \mid p \in AP \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathsf{X}\,\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

Cette syntaxe peut être découpée en deux parties, une partie gauche (les quatre premiers éléments) qui correspond à la logique propositionnelle classique, et une partie droite (les deux derniers éléments) qui correspond aux opérateurs modaux neXt et Until. L'opérateur neXt permet de spécifier une formule doit être vraie dans l'instant suivant (i.e., dans le successeur direct), tandis que l'opérateur Until exprime que la formule φ_1 doit être vraie jusqu'à ce que la formule φ_2 le soit.

Définition 6 – Relation de satisfaction

Soit $\rho = \alpha_1\alpha_2\alpha_3\ldots$ un ω-mot sur $\Sigma = 2^{AP}$, p une proposition atomique, et φ, φ_1, φ_2 des formules LTL. La relation de *satisfaction* $\rho \models \varphi$ est définie inductivement par :

$$\begin{array}{ll}
\rho \models \top & \\
\rho \models p & \text{si et seulement si } p \in \alpha_1 \\
\rho \models \neg\varphi & \text{si et seulement si } \neg(\rho \models \varphi) \\
\rho \models \varphi_1 \vee \varphi_2 & \text{si et seulement si } \rho \models \varphi_1 \text{ ou } \rho \models \varphi_2 \\
\rho \models \mathsf{X}\,\varphi_1 & \text{si et seulement si } \rho_{1\ldots} \models \varphi_1 \\
\rho \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{si et seulement si } \exists k \in \mathbb{N},\ \rho_{k\ldots} \models \varphi_2 \text{ et } \forall i \in \mathbb{N},\ 1 \leq i \leq k,\ \rho_i \models \varphi_1
\end{array}$$

Le sucre syntaxique pour les opérateurs $\wedge$ $\oplus$, $\leftrightarrow$ et $\rightarrow$ est défini exactement de la même manière que pour la section 1.1.1. Pour les opérateurs modaux, on définit aussi :

$$\mathsf{F}\,\varphi \stackrel{def}{=} \top \,\mathsf{U}\, \varphi \qquad\qquad \mathsf{G}\,\varphi \stackrel{def}{=} \bot \,\mathsf{R}\, \varphi$$

$$\varphi_1 \,\mathsf{R}\, \varphi_2 \stackrel{def}{=} \neg(\neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2) \qquad\qquad \varphi_1 \,\mathsf{W}\, \varphi_2 \stackrel{def}{=} \varphi_2 \,\mathsf{R}\, (\varphi_1 \vee \varphi_2)$$

Quatre nouveaux opérateurs font donc leur apparition : Finally et Globally (parfois notés respectivement $\Diamond$ et $\Box$), Release et WeakUntil. Le premier permet de s'assurer que le système atteindra « finalement » un état où la formule spécifiée sera vérifiée ; le second permet de vérifier qu'une propriété est « globalement » vraie pour tous les états de l'exécution ; le troisième permet de s'assurer qu'une formule sera continuement vraie jusqu'à un certain point et doit vérifier une autre formule jusqu'à ce point ; et enfin le denier permet de garantir qu'une formule est satisfaite jusqu'à ce qu'une autre formule le soit.

1. À l'exception du chapitre 7 qui utilise des heuristique dépendantes de LTL.

Définition 7 – Langage d'une formule LTL

Une formule LTL φ sur l'ensemble des propositions atomiques AP définit le langage :

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

Deux formules LTL φ_1 et φ_2 sont dites équivalentes si : $\mathscr{L}_{\varphi_1} = \mathscr{L}_{\varphi_2}$

Un système vérifie une propriété si et seulement si son langage est inclut dans celui de la propriété.

La définition précédente montre que langage d'une formule LTL correspond à l'ensemble des mots satisfaisant cette formule. Cette vision en terme de langage est intéressante car elle fait le lien avec les automates de Büchi utilisés pour la représentation des langages découlant des propriétés LTL. Les automates de Büchi permettent de décrire des exécutions (ω-mots) ne reconnaissent que les mots définis par le langage. Ainsi ils peuvent être vus comme des interpréteurs qui vont recevoir les exécutions de la structure de Kripke du modèle et vérifier si elles appartiennent ou non au langage qui a été défini.

1.2.2 Les automates de Büchi

Les automates de Büchi ont été introduits en 1962 par Julius Richard Büchi [13] comme méthode de décision pour la logique linéaire S1S. Il faut attendre que Moshe Vardi [87] introduise en 1986 l'*approche par automates pour la vérification* LTL afin qu'ils trouvent une place dans la vérification de systèmes au moyen de propriété LTL (cf. section 2.1). Dans cette approche, le système et la la propriété sont représentés au moyen d'automates de Büchi.

Un automate de Büchi permet de représenter un langage et offre une procédure de décision permettant de savoir si un ω-mot est reconnu ou non par ce langage. Les automates reconnaissant des ω-mots sont appelés *ω-automates*. Cette section introduit les ω-automates (automates de Büchi) qui seront manipulés dans ce mémoire et montre comment les structures de Kripke peuvent être considérées comme des ω-automates.

Les automates utilisés ici (Transition-based Generalized Büchi Automata ou TGBA) constituent une variante des automates de Büchi classiques (Büchi Automata ou BA). Ce choix est motivé à la fois parce que les algorithmes de traduction de formule LTL génèrent naturellement de tels automates [19] mais aussi parce que ces automates ont une représentation plus compacte que les BA tout en conservant la même expressivité. Les avantages de cette compacité seront détaillés dans le chapitre suivant.

Définition 8 – TGBA

Un *Automate de Büchi généralisé basé sur les transitions* (TGBA) $\mathcal{A} = \langle\, Q, q_0, AP, \mathcal{F}, \Delta \,\rangle$ est un LTS avec :

$\mathcal{F}$	: un ensemble non-vide de *marques d'acceptation* ;
$\Delta \subseteq Q \times 2^{AP} \times 2^{\mathcal{F}} \times Q$	: la relation de transition donnée sous la forme d'un quadruplet (état origine, minterm, ensemble d'acceptation, état destination).

Définition 9 – TBA

Un *Automate de Büchi basé sur les transitions* (TBA) $\mathcal{A} = \langle\, Q,\ q_0,\ AP,\ \mathcal{F},\ \Delta \,\rangle$ est un TGBA avec $|\mathcal{F}| = 1$

Les TGBA que nous considérons ici ont des ensembles d'acceptation non vides, i.e. $|\mathcal{F}| \geq 1$. Certaines définitions [22] autorisent $\mathcal{F} = \emptyset$: cela est équivalent à rajouter une marque d'acceptation sur toutes les transitions du TGBA. Un TGBA peut être vu comme un LTS dans lequel les transitions sont étiquetées par un ensemble de marques d'acceptation et un minterm. L'ensemble 2^{AP} peut être vu comme les actions du LTS, et une exécution d'un TGBA est définie de la même manière que pour un LTS en ignorant les marques d'acceptation portées sur les transitions.

Définition 10 – Langage d'un TGBA

Soit $\mathcal{A} = \langle\, Q,\ q_0,\ AP,\ \mathcal{F},\ \Delta \,\rangle$ un TGBA. Une *exécution acceptante* de $\mathcal{A}$ est une séquence infinie de transitions $\pi = (s_1, \alpha_1, f_1, d_1) \ldots (s_i, \alpha_i, f_i, d_i) \ldots$ avec $s_1 = q_0$, et $d_i = s_{i+1}$ qui visite toutes les marques d'acceptation infiniment souvent, i.e. :

$$\forall f \in \mathcal{F},\ \forall i \geq 1,\ \exists j \geq i,\ f_j \subseteq f$$

Un ω-mot $w = \rho_1\rho_2\cdots$ de $(2^{AP})^\omega$ est accepté par $\mathcal{A}$ s'il existe une exécution acceptante $\pi = (s_1, \alpha_1, f_1, d_1) \ldots (s_i, \alpha_i, f_i, d_i) \ldots$ telle que $\forall i, \rho_i = \alpha_i$. Le *langage* $\mathscr{L}_\mathcal{A} \subseteq (2^{AP})^\omega$ est l'ensemble des mots infinis acceptés par $\mathcal{A}$.

Définition 11 – Composante fortement connexe

Une *composante fortement connexe partielle* est un ensemble non vide d'états S tel que $S \subseteq Q$ et que $\forall\, s,\ s' \in Q, s \neq s' \implies s \rightsquigarrow s'$.

Une *composante fortement connexe* (notée SCC pour *Strongly Connected Component*) est l'ensemble maximal (au sens de l'inclusion) d'états formant une composante fortement connexe partielle.

Définition 12 – Composante fortement connexe acceptante et cycle acceptant

Soit $\mathcal{A} = \langle\, Q,\ q_0,\ AP,\ \mathcal{F},\ \Delta \,\rangle$ un TGBA et $S \subseteq Q$ une composante fortement connexe. La composante fortement connexe S est dite *acceptante* si et seulement si :

$$\forall f \in \mathcal{F}, \exists\, (q_1, \alpha_1, f_1, q_2) \in \Delta, \text{t.q. } q_1 \in S, q_2 \in S \text{ et } f_1 \subseteq f$$

Toute composante fortement connexe acceptante contient au moins un *cycle acceptant*, i.e un cycle qui visite chaque marque d'acceptation.

Si un TGBA possède une exécution acceptante, cela signifie qu'il possède au moins une composante fortement connexe acceptante accessible depuis l'état initial. Un automate avec une composante acceptante est dit *non-vide* et contient donc un cycle acceptant accessible depuis l'état initial. Par opposition, si l'automate ne possède pas de tel cycle il est dit *vide*.

Exemple.
La formule LTL « $\mathsf{G}(working \rightarrow \mathsf{F}\, waiting)$ » (traduction LTL de la formule de la page 23 : « Un étudiant qui travaille finit toujours par attendre ») peut être traduite par le TGBA de la figure 1.4. Pour plus de lisibilité (et dans tout le manuscrit) les transitions de cet automate ne sont pas étiquetées par des minterms mais par des formules propositionnelles. Il s'agit là d'une simplification d'écriture équivalente à répliquer la transition portant la formule propositionnelle pour chaque minterm de la disjonction de la formule. Cet automate n'est constitué que d'une unique composante fortement connexe qui contient aussi bien des exécutions acceptantes que des exécutions non-acceptantes. Par exemple l'exécution bouclant infiniment autour de l'état s_0 est acceptante tandis que celle commençant l'état s_0 et bouclant ensuite infiniment autour de l'état s_1 ne l'est pas.

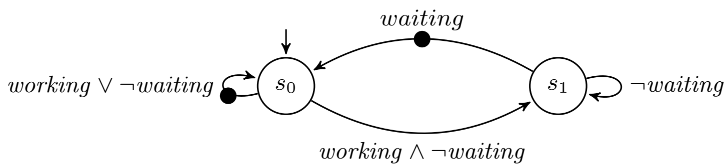


FIGURE 1.4 – TGBA pour la formule LTL « $\mathsf{G}(working \rightarrow \mathsf{F}\, waiting)$ » sur $AP = \{working, writing, idle, waiting\}$ et $\mathcal{F} = \{\bullet\}$.

Dans l'approche par automates pour le *model checking*, les structures de Kripke sont considérées comme des automates de Büchi : cela permet de ne manipuler qu'un unique formalisme tout au long de l'approche (cf. chapitre 2). L'idée sous-jacente est alors de considérer toutes les exécutions de la structure de Kripke comme acceptantes.

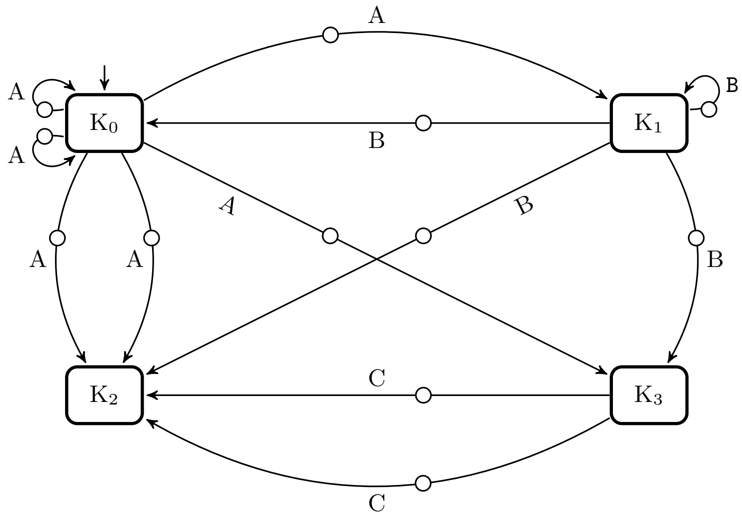
Définition 13 – Structure de Kripke vue comme un TGBA

Une structure de Kripke $\mathcal{K} = \langle\, Q,\ q_0,\ Act,\ \Delta,\ AP,\ L \,\rangle$ peut être transformée en un automate de Büchi $\mathcal{A} = \langle\, Q,\ q_0,\ AP,\ \mathcal{F},\ \Delta' \,\rangle$ reconnaissant le même langage, en posant :

$$\begin{array}{lcl} \mathcal{F} & = & \{\circ\} \\ \Delta' & : & (src, \alpha, dst) \in \Delta \implies (src, L(src), \circ, dst) \in \Delta \end{array}$$

La figure 1.5 présente l'automate résultant de la traduction de la structure de Kripke en suivant les règles définies ci-dessus. Toutes les transitions sont maintenant considérées comme acceptantes et les minterms ont simplement été transférés des états aux transitions. Par la suite les structures de Kripke seront manipulées comme des TGBA en utilisant la traduction ci-dessus.

Lors de la conversion d'une structure de Kripke en automate de Büchi, certains états n'ont pas de successeurs : il s'agit d'états bloquants. Comme les TGBA ne considèrent que les comportements infinis, ces états seront ignorés. Ces comportements bloquants peuvent néanmoins être préservés en les rendant infinis : il suffit de rajouter une boucle autour des états bloquants et de rajouter une proposition atomique indiquant cet état de blocage.



$$\begin{aligned} A &= \text{working} \wedge \neg\,\text{writing} \wedge \neg\,\text{idle} \wedge \neg\,\text{waiting} \\ B &= \neg\,\text{working} \wedge \text{writing} \wedge \neg\,\text{idle} \wedge \neg\,\text{waiting} \\ C &= \neg\,\text{working} \wedge \neg\,\text{writing} \wedge \neg\,\text{idle} \wedge \text{waiting} \end{aligned}$$

Figure 1.5 – $\mathcal{A}_{\mathcal{K}}$: traduction de la structure de Kripke de la figure 1.3 en TGBA. Les états ont été renommés pour plus de lisibilité : K_0 = ReadingOrThinking, K_1 = Writing, K_2 = Done, et K_3 = Waiting.

Exemple.
Dans l'exemple de la figure 1.3, une simple boucle autour de K_2 positionnant à $\top$ une proposition atomique *blocking* permet de considérer les états bloquants

1.3 Conclusion

Ce chapitre a introduit l'ensemble des définitions nécessaires à la compréhension de ce manuscrit. Il a été montré qu'un système pouvait être représenté au moyen d'une structure de Kripke dont la structure est de celle d'un LTS. L'ensemble des comportements possibles peut aussi être vu comme des séquences de mots sur le langage composé des actions du système.

Les automates de Büchi ont été introduits justement pour considérer des langages de mots infinis. Comme les systèmes que l'on étudie ont un nombre fini d'états mais des comportements infinis cette représentation convient parfaitement. La conversion d'une structure de Kripke en un automate de Büchi est naturelle puisque ces deux formalismes sont très proches.

De nombreuses variantes d'automates de Büchi ont été introduits depuis 1962, mais il apparaît que dans notre contexte les automates de Büchi généralisés sont particulièrement adaptés. Ces derniers, très concis, sont déjà utilisés dans le mécanisme de traduction d'une propriété (exprimée sous la forme d'une formule LTL) en un automate. La majorité des outils pour le model checking explicite utilisent néanmoins des automates non-généralisés (qui nécessitent une dégénéralisation) et l'utilisation d'automates généralisés dans le processus de vérification reste largement ignoré.

Dans ce manuscrit nous nous focalisons sur le processus de vérification au travers d'automates de Büchi généralisés. Cette approche travaille directement sur les automates et n'est donc pas cantonnée à une logique temporelle linéaire particulière. En effet, de nombreuses logiques temporelles linéaires existent avec des pouvoir d'expression différents. Comme toutes ces logiques peuvent être traduite en automates de Büchi, nos travaux y sont directement applicables.

Chapitre 2

L'équité dans l'approche par automates pour le *model checking*

You can prove anything you
want by coldly logical reason —
if you pick the proper postulates.

Isaac Asimov

Sommaire

Ce chapitre introduit l'approche par automates pour le model checking*. Nous nous intéressons particulièrement ici à l'équité qui justifie en partie l'utilisation des automates généralisés comme cadre d'étude de cette thèse.*

2.1 Approche par automates pour le *model checking*

L'*approche par automates pour le model checking* a été introduite par Vardi [87] pour fournir un canevas automatisé, unifié, et extensible au processus de vérification.

2.1.1 Détails de l'approche

Dans cette approche, l'automate de Büchi de la propriété est considéré comme un interpréteur qui va valider les exécutions de la structure de Kripke représentant le système. Cette représentation offre de nombreux avantages : (1) elle permet de réduire le problème du *model checking* à

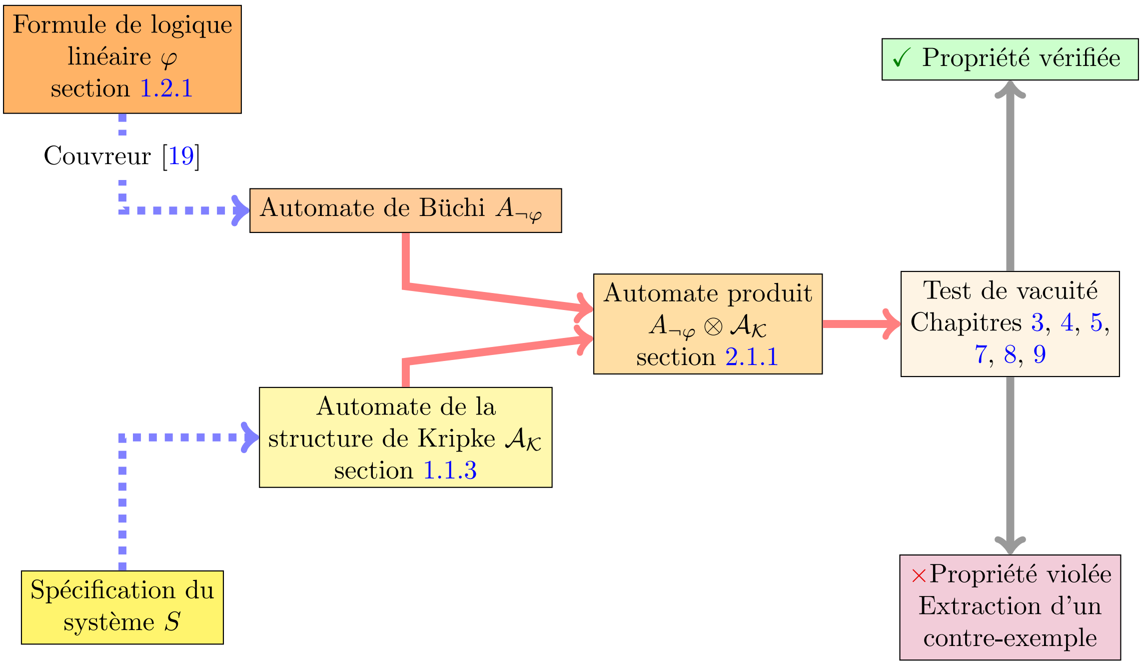


FIGURE 2.1 – Approche par automates pour le *model checking*

un pur problème de théorie des automates, (2) elle peut aisément être étendue à d'autre types d'automates, (3) elle offre un mécanisme de vérification indépendant de la propriété à vérifier et (4) elle est indépendante de la logique à temps linéaire utilisée.

Cette approche teste l'inclusion du langage du système dans celui de la propriété. Si cette inclusion existe la propriété est dite *vérifiée*, sinon elle est dite *violée* et il existe un *contre-exemple*, i.e. une exécution du système qui invalide la propriété.

Définition 14

Soit $\mathcal{K}$ une structure de Kripke définissant le langage $\mathscr{L}_{\mathcal{K}}$ et φ une formule de logique temporelle à temps linéaire définissant le langage $\mathscr{L}_{\varphi}$. Dire que $\mathcal{K}$ « satisfait » φ peut s'exprimer de la manière suivante :

$$\begin{array}{llll} \mathcal{K} \models \varphi & \Leftrightarrow & \mathscr{L}_{\mathcal{K}} & \subseteq \mathscr{L}_{\varphi} \\ & \Leftrightarrow & \mathscr{L}_{\mathcal{K}} \cap \overline{\mathscr{L}_{\varphi}} & = \emptyset \\ & \Leftrightarrow & \mathscr{L}_{\mathcal{K}} \cap \mathscr{L}_{\neg\varphi} & = \emptyset \end{array}$$

Comme les automates de Büchi représentent des langages, le test d'inclusion des langages peut être réalisé en travaillant directement sur des automates. Cette technique est présenté figure 2.1 : les flèches représentent des étapes de traduction et le sens de lecture est de gauche à droite.

Dans ce schéma le modèle est converti en une structure de Kripke, tandis que la négation de la formule de logique temporelle linéaire est convertie en automate de Büchi. Nier la formule de logique temporelle avant de la traduire permet d'éviter une complémentation de l'automate de Büchi qui est une opération très coûteuse. Ces deux automates sont ensuite combinés en un *automate produit*.

Définition 15 – Produit synchronisé

Soient deux TGBA $\mathcal{A}$ et $\mathcal{A}'$ tels que $\mathcal{A} = \langle Q,\, q_0,\, AP,\, \mathcal{F},\, \Delta \rangle$ $\mathcal{A}' = \langle Q',\, q_0{}',\, AP,\, \mathcal{F}',\, \Delta' \rangle$ sur le même ensemble AP de propositions atomiques et tels que $\mathcal{F} \cap \mathcal{F}' = \emptyset$.

Le *produit synchronisé* de $\mathcal{A}$ et $\mathcal{A}'$ noté $\mathcal{A} \otimes \mathcal{A}'$ est défini par $\mathcal{A}'' = \langle Q'',\, q_0{}'',\, AP,\, \mathcal{F}'',\, \Delta'' \rangle$ tel que :

$$
\begin{array}{lll}
Q'' & = Q \times Q' & : \text{l'ensemble des états ;} \\
q_0{}'' & = (q_0, q_0{}') & : \text{l'état initial est composé des deux états initiaux ;} \\
\mathcal{F}'' & = \mathcal{F} \cup \mathcal{F}' & : \text{l'union des ensembles d'acceptations.} \\
\Delta'' & = \{((s,s'), \alpha, f \cup f', (d,d')) & \mid (s,\alpha,f,d) \in \Delta \text{ et } (s',\alpha,f',d') \in \Delta'\} \\
 & & : \text{les transitions « synchronisées » des deux automates.}
\end{array}
$$

Le langage $\mathscr{L}_{\mathcal{A}\otimes\mathcal{A}'}$ de l'*automate produit* est par construction :

$$\mathscr{L}_{\mathcal{A}\otimes\mathcal{A}'} = \mathscr{L}_{\mathcal{A}} \cap \mathscr{L}_{\mathcal{A}'}$$

Par la suite, pour $q = (s, s') \in Q''$ la *projection* d'un état du produit sur l'automate $\mathcal{A}$ est notée : $\mathcal{P}_{|\mathcal{A}}(q) = s$. De manière similaire $\mathcal{P}_{|\mathcal{A}'}(q) = s'$.

Le *test de vacuité* (ou *emptiness check*) permet de tester si le langage d'un automate est vide, i.e. s'il ne contient pas de cycles acceptants. Lorsque le test de vacuité détecte un cycle acceptant ou une composante fortement connexe acceptante, il peut retourner un contre-exemple, c'est-à-dire une exécution invalidant la propriété.

2.1.2 Cas pratique

Exemple.
Reprenons l'exemple d'une classe d'Archimède composée d'un unique étudiant (présenté au chapitre précédent). On souhaite toujours vérifier la propriété « Un étudiant qui travaille finit toujours par attendre », qui s'exprime par la formule LTL $\varphi = \mathsf{G}(working \rightarrow \mathsf{F}\, waiting)$. Comme montré par le schéma de la figure 2.1, cette formule doit tout d'abord être niée :

$$
\begin{aligned}
\neg\varphi &= \neg\,(\mathsf{G}(working \rightarrow \mathsf{F}\, waiting)) \\
&= \mathsf{F}\,(working \wedge \mathsf{G}\,\neg\, waiting)
\end{aligned}
$$

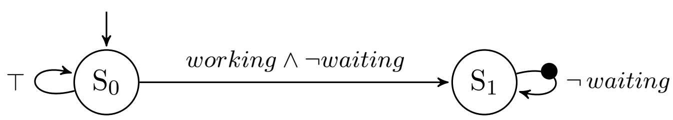


FIGURE 2.2 – TGBA la formule LTL « $\varphi = \mathsf{G}(working \rightarrow \mathsf{F}\, waiting)$ ». $\mathcal{F} = \{\bullet\}$.

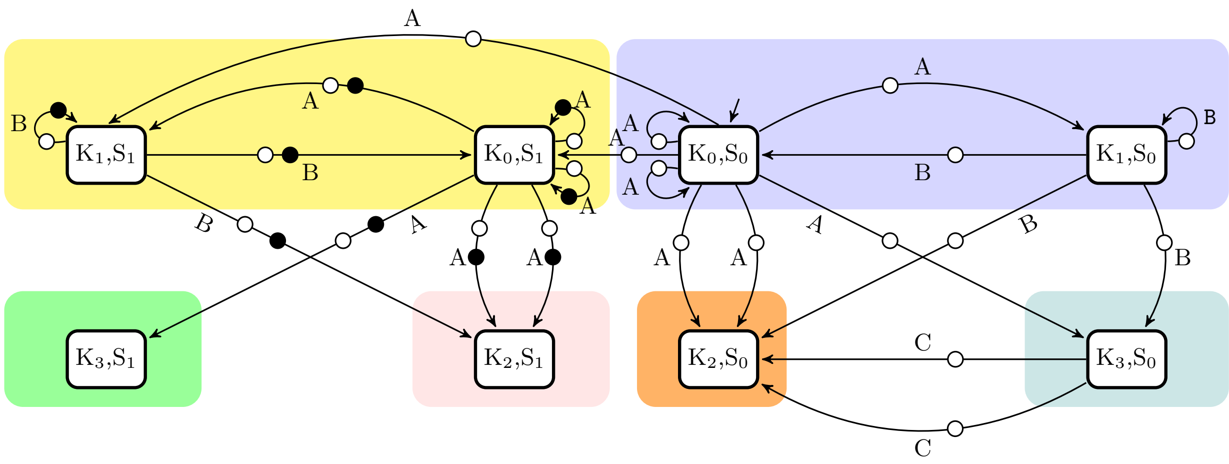


FIGURE 2.3 – Produit synchronisé des TGBA des figures 1.5 et 2.2. La légende pour les étiquettes est la même que pour la figure 1.5. $\mathcal{F} = \{\bullet, \circ\}$ ($\bullet$ vient de $\mathcal{A}_{\neg\varphi}$ et $\circ$ de $\mathcal{A}_{\mathcal{K}}$).

La figure 2.2 présente $\mathcal{A}_{\neg\varphi}$ le TGBA reconnaissant le langage $\mathscr{L}_{\neg\varphi}$ tandis que la figure 1.5 (chapitre précédent) présente $\mathcal{A}_{\mathcal{K}}$ le TGBA issu de la structure de Kripke modélisant un étudiant de la classe d'Archimède. Une fois $\mathcal{A}_{\mathcal{K}}$ et $\mathcal{A}_{\neg\varphi}$ construits le produit synchronisé peut être réalisé comme présenté figure 2.3. Le tableau suivant récapitule les informations sur ces trois automates :

	$\mathcal{A}_{\mathcal{K}}$ (Fig. 1.5)	$\mathcal{A}_{\neg\varphi}$ (Fig. 2.2)	$\mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\neg\varphi}$ (Fig. 2.3)
Nombre d'états	4	2	8
Nombre de transitions	12	3	23
Nombre de composantes fortement connexes	3	2	6

L'automate du produit $\mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\neg\varphi}$ possède deux fois plus d'états que l'automate du système : il s'agit du pire cas pour les états accessibles puisque la taille de ce produit est de $|\mathcal{A}_{\mathcal{K}}| \times |\mathcal{A}_{\neg\varphi}|$. Le nombre de composantes fortement connexes est lui aussi impacté et six composantes apparaissent figure 2.3 là où $\mathcal{A}_{\mathcal{K}}$ n'en possède que trois. L'analyse de ces composantes est importante car elles peuvent contenir des cycles acceptants. Dans cet exemple, un cycle acceptant doit visiter infiniment souvent les marques $\bullet$ et $\circ$. On constate que seule la composante jaune (composée des états (K_0,S_1) et (K_1,S_1)) peut contenir de tels cycles et l'exécution (K_0,S_0) $\rightarrow$ (K_1,S_1) $\rightarrow$ (K_1,S_1) $\rightarrow \ldots$ constitue un des contre-exemples possibles : la propriété n'est donc pas vérifiée. En revanche, l'exécution (K_0,S_0) $\rightarrow$ (K_0,S_0) $\rightarrow \ldots$ n'est pas un contre-exemple puisqu'elle ne visite pas toutes les marques d'acceptation de $\mathcal{F}$.

2.2 Équité

Le seul système considéré jusqu'à présent n'est composé que d'un unique processus (l'étudiant), mais des systèmes résultants de l'entrelacement des exécutions de plusieurs processus, existent. Dans ce cas, il est difficile d'assurer l'*équité*, e.g. qu'à n'importe quel instant un processus non bloqué a l'assurance de devenir actif ou de pouvoir progresser dans le futur.

2.2.1 Différentes formes d'équité

L'exemple de la section précédente considère une classe composée d'un unique étudiant et les transitions étiquettées par `giveup`, `abort` et `solutionfound`, qui synchronisent plusieurs étudiants, n'ont alors pas d'utilité. Dans le chapitre précédent, nous avons vu comment ces actions se synchronisent pour construire le modèle associé à une classe de deux étudiants (figure 1.2 page 20). Parmi les exécutions possibles, certaines ne sont pas *équitables* car elles ne permettent pas l'exécution de chaque processus : l'exécution $\texttt{write}_1$,$\texttt{think}_1$,$\texttt{write}_1$,$\texttt{think}_1 \ldots$ [1] laisse le second étudiant dans l'état ReadingOrThinking par exemple.

Dans un système réel, il est peu probable que, de telles exécutions existent puisque les processus sont généralement ordonnancés de manière équitable. Ces exécutions non équitables peuvent conduire à des résultats erronés lors de la vérification de propriétés de logique temporelle. Une façon de palier cela est de modéliser l'ordonnanceur qui va « donner la main » régulièrement à chaque processus et donc supprimer les comportements irréalistes. Cette modélisation peut s'avérer fastidieuse et va surtout accroître l'explosion combinatoire. Une autre approche peut être envisagée en effectuant la vérification sous *hypothèses d'équité*. Dans un système multi-processus ces hypothèses peuvent être définies de la manière suivante :

Équité Inconditionnelle ($\mathcal{E}_i$) : spécifie qu'un processus pourra *progresser* infiniment souvent ;

Équité Faible ($\mathcal{E}_w$) : spécifie que tous les processus qui seront *actifs* infiniment souvent pourront *progresser* infiniment souvent ;

Équité Forte ($\mathcal{E}_s$) : spécifie que tous les processus qui seront *actifs* continuement à partir d'un certain point *progresserons* infiniment souvent.

Un processus est dit *actif* s'il est dans un état où il peut exécuter une transition [2] ; un processus est dit en *progression* s'il exécute une transition. Les hypothèses d'équité vont en ordre croissant de contraintes, et si $\mathrm{E}_{\mathcal{E}_i}$ (resp. $\mathrm{E}_{\mathcal{E}_w}$, resp. $\mathrm{E}_{\mathcal{E}_s}$) représente l'ensemble des propriété exprimable avec l'équité inconditionnelle (resp. faible, resp. forte) on a :

$$E_{\mathcal{E}_i} = E_{\mathcal{E}_w} \subseteq E_{\mathcal{E}_s}$$

2.2.2 Gestion de l'équité faible et inconditionnelle

La notion d'équité peut être exprimée avec un automate de Büchi, mais l'équité faible et inconditionnelle peuvent être traduites directement au moyen de structures de Kripke équitables.

Définition 16 – Structure de Kripke équitable

Une *structure de Kripke équitable* $\mathcal{K} = \langle\, Q,\, q_0,\, Act,\, \Delta,\, AP,\, L,\, \mathcal{F} \,\rangle$ est une structure de Kripke dans laquelle la fonction de transition a été étendue pour stocker des marques d'acceptation, i.e. :

$\Delta \subseteq Q \times Act \times 2^{\mathcal{F}} \times Q$: la relation de transition donnée sous la forme d'un quadruple (état origine, action, ensemble d'acceptation, état destination) appelés *transitions* ;

$\mathcal{F}$: un ensemble de marques d'acceptation.

1. Nous utilisons ici les actions portées par les transitions pour décrire une exécution pour plus de lisibilité.
2. Dans l'exécution $\texttt{write}_1$,$\texttt{think}_1$,$\texttt{write}_1$,$\texttt{think}_1\ldots$ le second étudiant est toujours actif puisqu'il a la possibilité de passer dans l'état Writing par exemple. Lors de l'utilisation d'une table de composition cela n'est pas toujours le cas. Par exemple, l'exécution $\texttt{giveup}_2$,$\texttt{write}_1$,$\texttt{think}_1$,$\texttt{write}_1$,$\texttt{think}_1\ldots$ laisse le second étudiant dans un état où il n'est pas actif.

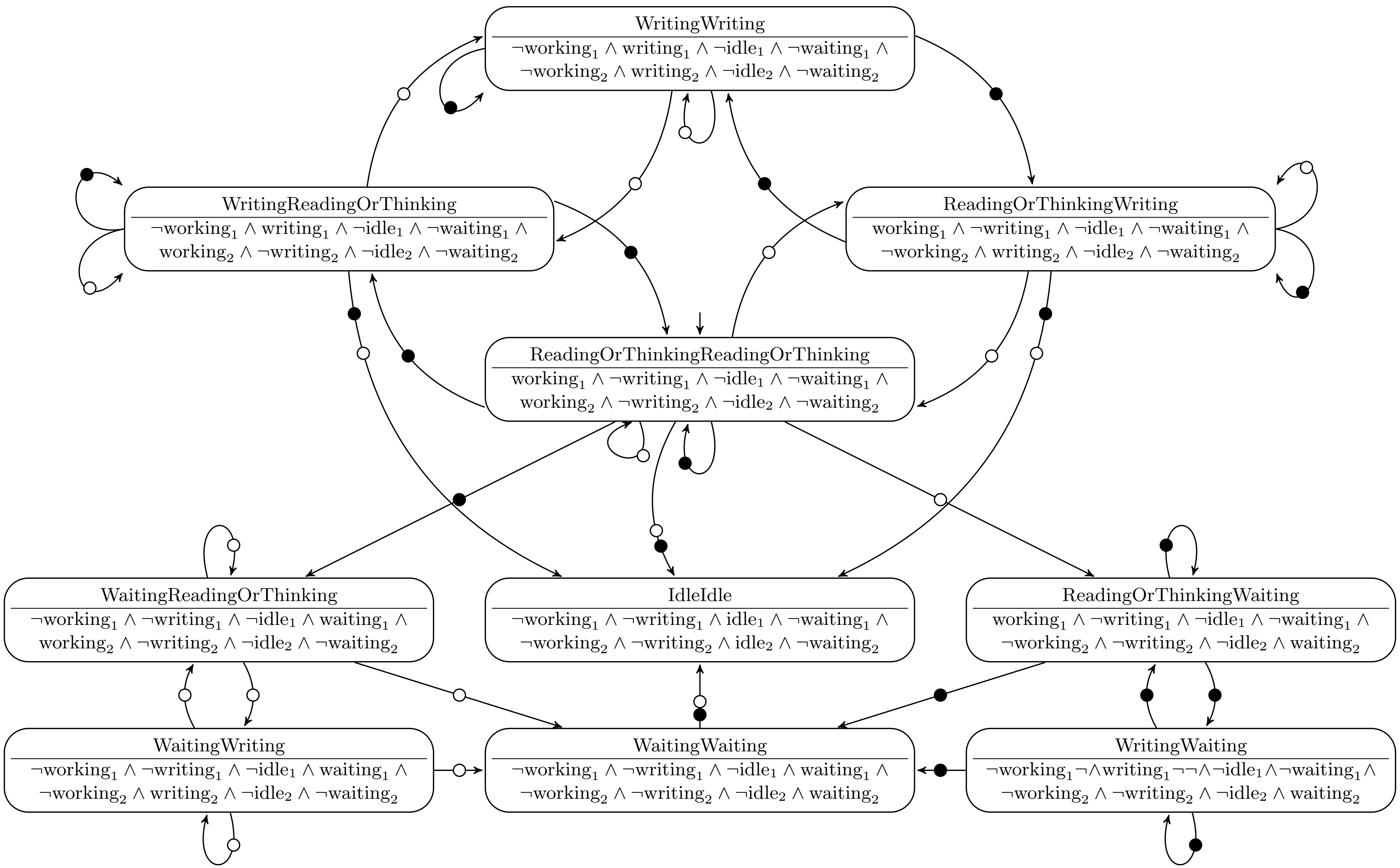


Figure 2.4 – Structure de Kripke équitable pour un système composé de deux étudiants.

Les structures de Kripke équitables sont très proches des TGBA puisqu'une exécution est acceptante si et seulement si toutes les marques d'acceptation sont présentes infiniment souvent. La conversion en automate de Büchi généralisé (TGBA) est alors immédiate.

Exemple.
La figure 2.4 présente la transformation en structure équitable du LTS présenté figure 1.2, en faisant l'hypothèse que l'ordonnanceur est équitable. Toutes les transitions du premier étudiant sont étiquetées par ● tandis que celles du second sont étiquetées par ○. Comme chaque processus possède sa propre marque d'acceptation il s'agit d'une équité faible et, les seuls contre-exemples qui peuvent être détectés sont ceux « équitables ». Les comportements non réalistes sont alors ignorés par le test de vacuité et l'exécution ReadingOrThinkingReadingOrThinking → ReadingOrThinkingWaiting → WritingWaiting → ReadingOrThinkingWaiting → ... ne peut jamais faire partie un contre-exemple car la marque ○ n'est jamais vue. Ici, toutes les transitions portent des marques d'acceptation mais seules certaines transitions peuvent les porter pour s'assurer d'une progression spécifique.

2.2.3 Gestion de l'équité par l'intermédiaire des automates de Büchi

L'équité forte n'est pas exprimable avec les structures de Kripke équitables présentées ci-dessus, mais elle l'est au travers de la logique temporelle. Vérifier une formule φ sous une hypothèse d'équité ψ se traduit par la vérification de la formule $\psi \rightarrow \varphi$. Les différentes formes d'équité s'expriment de la manière suivante (avec a et b deux propositions atomiques du système, telle que a dénote qu'un processus est actif et b dénote qu'il progresse) :

$$
\begin{aligned}
\mathcal{E}_i &\equiv \mathsf{G}\,\mathsf{F}\,a \\
\mathcal{E}_w &\equiv \mathsf{F}\,\mathsf{G}\,a \rightarrow \mathsf{G}\,\mathsf{F}\,b \\
\mathcal{E}_s &\equiv \mathsf{G}\,\mathsf{F}\,a \rightarrow \mathsf{G}\,\mathsf{F}\,b
\end{aligned}
$$

La figure 2.5 présente les TGBA issus de la traduction des formules LTL représentant les trois formes d'équité. L'équité faible et inconditionnelle sont représentables par un automate à un unique état et deux transitions tandis que l'équité forte est représentée par un automate à trois états et six transitions. Lorsque l'on souhaite combiner plusieurs hypothèses d'équité il suffit de faire un « et logique » entre les formules : lorsque les automates n'ont qu'un seul état la combinaison des hypothèses ne va affecter que le nombre de transitions, sinon le nombre d'états est lui aussi affecté.

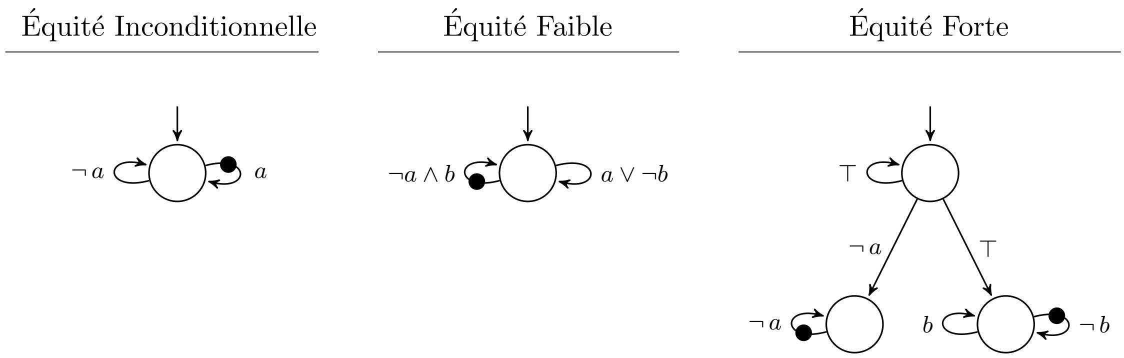


FIGURE 2.5 – Expression de l'équité au moyen d'un automate

Effectuer la vérification d'une propriété sous une hypothèse d'équité ne change pas la procédure des vérification puisque :

$$\begin{aligned}\text{Vérifier } \varphi \text{ sous l'hypothèse } \psi &\equiv \mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\neg(\psi\rightarrow\varphi)} \\ &\equiv \mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\psi\wedge\neg\varphi} \\ &\equiv \mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\psi} \otimes \mathcal{A}_{\neg\varphi}\end{aligned}$$

Comme le produit synchronisé est commutatif[3], il est équivalent de synchroniser d'abord le système avec l'hypothèse d'équité ou d'effectuer la synchronisation classique entre le système et la propriété avant de rajouter les contraintes d'équité. L'ordre de composition est néanmoins important car il impacte la taille de l'automate résultant. Par exemple, l'équité inconditionnelle et faible peuvent être représentées au moyen d'un automate à état ce qui n'impacte pas la taille de l'automate du produit synchronisé à l'inverse des hypothèses d'équité forte. Afin d'être homogène et de ne pas avoir à modifier le modèle pour y rajouter des hypothèses d'équité, nous préférerons dans ce manuscrit l'expression de l'équité au travers des formules de logiques temporelles.

Note : qu'il existe d'autres types d'automates (Streett, Rabin, ...) permettant une expression plus concise des hypothèses d'équité forte. Néanmoins, ces automates repoussent la complexité vers les tests de vacuité et ne sont pas naturellement utilisés dans le cadre de la traduction de formules de logique temporelle. Nous ne les considérerons donc pas dans ce manuscrit.

2.3 Test de vacuité

Cette section introduit les algorithmes testant la vacuité et indique lesquels sont compatibles avec les structures de Kripke équitables. Un tel test permet de détecter si le langage d'un automate de Büchi est vide ou non : cela n'est possible que par une exploration (totale si son langage est vide) de l'automate. *Explorer* un automate implique de visiter et traiter ses états dans un ordre systématique. *Visiter* un état signifie le découvrir tandis que le *traiter* signifie effectuer tous les traitements qui lui sont liés (typiquement visiter tous ses successeurs directs). Il existe de nombreux parcours mais les deux suivants sont les plus classiques :

Depth-First Search (DFS) : le parcours est fait en « profondeur », les successeurs d'un états s sont traités avant de traiter s. Pour l'automate de la figure 2.6 un ordre de visite peut être : A,B,E,G,F,C,D tandis que l'ordre de traitement postfixe associé peut être : G,E,F,B,D,C,A.

Breadth-First Search (BFS) : le parcours est fait en « largeur » c'est-à-dire que les états sont visités par niveaux. L'ordre de parcours est identique à l'ordre de traitement : pour l'automate de la figure 2.6 cet ordre est : A,B,C,E,F,D,G.

La complexité temporelle de ces parcours est linéaire par rapport au nombre d'arcs, et les états visités sont sauvegardés pour éviter de les visiter deux fois. De plus, chaque parcours doit maintenir un ensemble d'états à traiter : cet ensemble est généralement plus petit pour les parcours DFS que pour les parcours BFS.

3. Le produit synchronisé est commutatif dans le sens ou il produit toujours un automate reconnaissant le même langage.

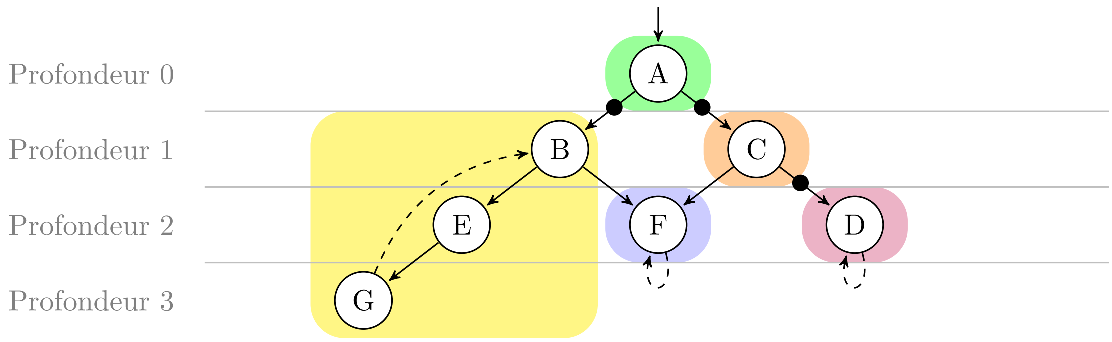


Figure 2.6 – Exemple d'automate avec 5 SCCs. Découpage par profondeur. Les arcs en pointillés sont des *arcs fermants* et le premier successeur d'un état est son fils gauche.

Les tests de vacuité basés sur des DFS donc sont plus adaptés au *model checking* explicite séquentiel car : (1) la profondeur des automates manipulés est généralement plus petites que leur largeur, (2) pour un niveaux donné il n'est pas nécessaire de stocker tous les successeurs, et (3) elles permettent de détecter les arc fermants. Un *arc fermant* est un arc allant d'un état vers un de ses prédécesseurs (directs ou indirects), il ferme donc un cycle qui peut contenir une exécution acceptante. Notons néanmoins que les tests de vacuité basés sur des BFS facilitent l'extraction de contre-exemples minimaux.

Test de vacuité basés sur l'énumération des composantes fortement connexes L'idée d'énumérer les composantes fortement connexes a été proposée en 1985 par Lichtenstein et Pnueli [59] dans une approche de tableaux sémantiques comme preuve de satisfiabilité d'une formule. L'approche s'intéresse aux composantes « terminales » de l'automate du modèle, i.e. les composantes sans transitions sortantes. Comme tout automate avec un ensemble non-vide d'états en possède au moins une, l'idée est de chercher ces composantes fortement connexes pour vérifier si elles invalident la propriété. Les états des composantes « terminales » sont ensuite supprimés et la procédure est relancée. La recherche s'arrête dès qu'une composante invalidante est trouvée ou bien dès que l'automate de la structure de Kripke n'a plus d'états. Cet algorithme est inefficace puisque la recherche de composantes « terminales » peut être effectué N fois, où N est le nombre de composantes fortement connexes.

Nous avons vu qu'il était néanmoins possible de rechercher directement les composantes acceptantes de l'automate du produit. Il existe des algorithmes efficaces pour énumérer les composantes fortement connexes [21, 78, 82]. Nous ne nous intéressons ici qu'aux algorithmes de Tarjan [82] et Dijkstra [21] : l'algorithme de Sharir/Kosaraju [78] n'est pas adapté au *model checking* car il requiert une exploration arrière des transitions (à cause d'une génération « à la volée », détails section 2.4). Les algorithmes de Tarjan et de Dijkstra ont un fonctionnement similaire : ils utilisent un parcours DFS pour l'exploration et détectent une composante fortement connexe uniquement lorsque tous les successeurs de la *racine* ont été visité. La *racine* d'une composante fortement connexe est le premier de ses états rencontré par le DFS. Les racines de l'automate de la figure 2.6 sont : A, B, C, F, et D. Enfin, ces algorithmes doivent maintenir les ensembles de marques d'acceptation présents pour chaque composante fortement connexe ce qui peut être coûteux en mémoire.

Tests de vacuité basés sur la recherche de cycles acceptants En 1991, Courcoubetis et al. [18] ont proposé de transposer le problème de recherche de composantes acceptantes en

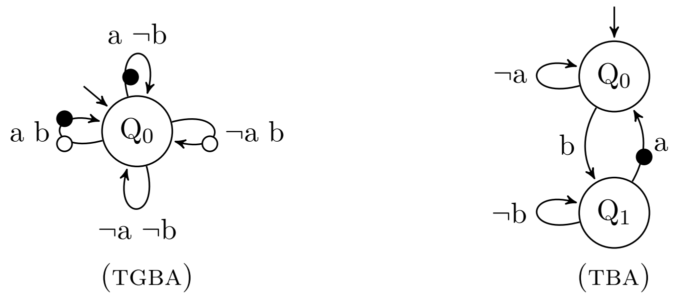


Figure 2.7 – Dégénéralisation d'un TGBA (à gauche) en un TBA équivalent.

un problème de recherche de cycle acceptants pour réduire la consommation mémoire des algorithmes. L'idée originale (présentée sur les BA) est d'utiliser un premier DFS qui va ordonner les états acceptants selon leur ordre *postfixe* (ordre de traitement induit par le DFS). Un second DFS est ensuite utilisé pour chercher un cycle autour de ces états acceptants [4]. Il a été montré que l'utilisation de l'ordre postfixe permet de ne revisiter au maximum qu'une fois tous les états. Courcoubetis et al. [18] examinent aussi la possibilité d'imbriquer ces deux parcours au moyen d'un NDFS (*Nested Depth First Search*) : dès qu'un état acceptant est traité, une recherche de cycle autour de cet état est faite. Si l'on applique cette idée aux TBAà l'automate de la figure 2.6 des parcours imbriqués seront donc lancés dès que tous les successeurs (directs ou indirects) des transitions (A, B), (A, C), et (C, D) auront été visités.

Certains tests de vacuité ne supportent que des TGBA avec une unique marque d'acceptation, i.e. $|\mathcal{F}| = 1$. Pour pouvoir tester la vacuité d'un TGBA avec ces algorithmes, une *dégénéralisation* est requise. Cette opération transforme un automate généralisé en un automate avec $|\mathcal{F}| = 1$, i.e. un TBA [5]. Bien que les langages résultants soient identiques la dégénéralisation d'un TGBA avec n états peut conduire à la création d'un TBA avec $n \times |\mathcal{F}|$ états. La figure 2.7 montre l'impact en terme d'états et transitions de la dégénéralisation d'un TGBA. Dans cet exemple, le TGBA à un état et quatre transitions est converti en un TBA avec deux états et quatre transitions. Dans un tel cas, doubler la taille de l'automate conduit à doubler la taille du produit. L'utilisation d'automates généralisés permet donc de réduire l'explosion combinatoire.

Note : Les tests de vacuité explorent l'automate en considérant uniquement les marques d'acceptation : les étiquettes portées par les transitions ne leur sont d'aucune utilité. Par la suite elles seront donc omises quand cela est possible.

2.4 Combattre l'explosion combinatoire

L'utilisation d'automates de Büchi généralisés permet de combattre l'explosion combinatoire par plusieurs moyens : (1) l'automate de la propriété étant plus petit son produit avec l'automate de la structure de Kripke l'est aussi, et (2) l'utilisation de structures de Kripke équitable est possible ce qui peut éviter une synchronisation avec un automate représentant l'équité. Il est donc nécessaire de travailler avec des tests de vacuité supportant de tels automates.

4. Les BA portent les marques d'acceptation sur les états.

5. L'approche proposée par Vardi [87] travaille sur les automates de Büchi (BA). Ces automates ne sont pas généralisés et la marque d'acceptation est portée par les états. Ces automates étant moins concis que les TBA ce sont ces derniers que nous préférerons lorsque l'utilisation d'automates non généralisés est requise.

D'autres techniques de réductions existent cependant et ont été présentées par Valmari [86], cette section en énumère les principales.

On-the-fly L'approche de la figure 2.1 nécessite de construire l'automate de la propriété et l'automate du système pour pouvoir ensuite tester la vacuité du produit de ces deux automates. Gerth et al. [36] constatent que l'introduction d'un bug dans un système correct peut faire exploser l'espace d'état de l'automate du produit puisqu'il n'obéit plus à des règles strictes. Par exemple, une erreur dans un protocole de communication peut permettre à un processus de progresser sans avoir reçu d'acquittement. Tous les états du système doivent alors intégrer cette progression erronée ce qui peut augmenter significativement le nombre d'états et de transitions dans le produit.

Pour palier cela Gerth et al. [36] proposent de construire l'espace d'état du produit « à la demande » : lors de l'exploration de l'automate du produit par le test de vacuité, chaque nouvel état est construit dynamiquement. En cas de présence d'un bug, seule la partie du produit synchronisé ayant conduit à sa détection est construite, évitant peut être une explosion combinatoire. Dans le cas où le système est correct, cette méthode n'introduit pas de surcoûts. L'intérêt n'existe que si l'algorithme testant la vacuité est capable de détecter un contre-exemple avant de construire l'intégralité de l'espace d'état.

Notons que dans la pratique, l'automate de la propriété est petit au regard de l'automate du système. Le construire par avance permet de le simplifier ce qui aide à combattre l'explosion combinatoire.

Bit State Hashing Dans le cas ou la propriété est vérifiée, l'intégralité de l'espace d'état du produit doit être exploré : il faut le conserver en mémoire (disque ou RAM). En 1987, Holzmann [43] exploite le fait que certains systèmes sont trop « gros » pour tenir en mémoire. Si un automate possède N états de taille s (bits), la mémoire minimale requise pour stocker l'intégralité de cet automate est $N \times s$. Si la mémoire disponible est strictement inférieur à cette limite, l'analyse de ce système n'est pas possible.

La technique présentée par Holzmann [43] repose sur un hachage des états. L'idée est de compresser l'espace d'état dans un tableau T de taille m. Initialement, tous les bits de T sont positionnés à $\perp$. Lorsqu'un état est découvert, une fonction de hachage va être appliquée sur cet état. Cette fonction retourne un entier i compris entre 0 et m. Il suffit alors de regarder si $T[i] = \perp$ pour savoir si cet état a déjà été visité ; si ce n'est pas le cas alors $T[i] \leftarrow \top$. Le risque de collision peut être atténué via l'utilisation de plusieurs fonctions de hachage indiçant soit le même tableau (multi-hachage séquentiel) soit des tableaux différents (multi-hachage).

Le test de vacuité devient alors une procédure de semi-décision puisque certains états du systèmes ont pu être ignorés à cause des collision dans T. Néanmoins trois points sont à noter : (1) les contre-exemples détectés en sont vraiment au regard du système, (2) lorsqu'aucun contre-exemple n'est trouvé on peut affirmer la validité d'un système que l'on n'aurait pas pu vérifier sinon (faute de mémoire) (3) cette technique est efficace lors de la recherche de bugs puisqu'il suffit d'augmenter régulièrement la taille des tables de hachage.

State Space Caching Cette idée a été introduite en 1985 par Holzmann [42] et raffinée en 1992 par Godefroid et al. [39]. Contrairement à la technique précédente qui peut conduire à une approximation de la validité d'une formule, le *State Space Caching* est une technique exacte mais qui peut conduire à de la redondance de travail. Lorsque tous les successeurs d'un états ont été visités durant un DFS, cet état doit être conservé jusqu'à la fin du parcours pour éviter d'avoir à le reparcourir. Cette conservation peut avoir un impact non négligeable sur la mémoire utilisée.

L'idée de Holzmann [42] est de ne conserver qu'un certain nombre d'états. Dès que cette limite est atteinte, les nouveaux états remplaceront les plus anciens. Ainsi, la vérification est juste mais certains états pourront être traités plusieurs fois ce qui ralenti la vérification.

L'application de cette technique est néanmoins limitée et il a été montré qu'au delà d'un certain point de réduction de la zone de stockage les temps d'exécutions des tests de vacuité deviennent trop coûteux.

Les deux dernières techniques ont été combinées à maintes reprises [39, 81] particulièrement avec des techniques d'ordre partiels [38] qui utilisent la commutativité du modèle (typiquement la composition des processus) pour réduire l'espace d'état devant être exploré. Ainsi certaines exécutions équivalentes de l'automate du produit peuvent être ignorées. Duret-Lutz et al. [23] proposent de construire un produit particulier permettant d'agréger les état, Holzmann [45] propose l'utilisation d'une compression des états ou des chemins pour maximiser l'utilisation de la mémoire. Ces différentes approches ont majoritairement été étudiées dans le cadre des algorithmes (non généralisés) basés sur la détection des cycles acceptants.

2.5 Conclusion

Dans ce chapitre nous avons vu que l'expression des propriétés permet d'éviter la détection de contre-exemples irréalistes sur des systèmes complexes. De plus, nous avons vu que l'utilisation d'automates généralisés contribuent à la réduction de l'explosion combinatoire.

Cette explosion peut être également évitée lors du test de vacuité par une exploration « à la volée » qui ne construit que la sous-partie du produit synchronisé nécessaire à la détection de contre-exemples. Cette technique permet de réduire la mémoire utilisée lorsqu'il existe un contre-exemple sans pour autant engendrer de surcoûts lorsque la propriété est vérifiée.

Le *Bit State Hashing* permet l'analyse de systèmes trop volumineux pour tenir en mémoire, tandis que le *State Space Caching* permet de fixer une borne mémoire en augmentant la redondance de travail effectué. Ces deux techniques combinées à la présence de mémoire de plus en plus grande sur le marché aujourd'hui tendent à déporter la limitation sur le temps d'exécution plutôt que sur la mémoire (comme c'était le cas dans les années 1990). L'amélioration du temps d'exécution pour les algorithmes testant la vacuité et leur combinaison avec les techniques de la réduction de la mémoire ont été massivement étudiés pour une seule des familles d'algorithme qui ne gère pas naturellement l'équité.

Dans la suite de ce manuscrit nous verrons comment combiner ces approches pour améliorer les temps d'exécution nécessaires à la vérification de propriété de logiques temporelles.

Deuxième partie

Contributions aux tests de vacuité séquentiels

Chapitre 3

Les tests de vacuité pour les automates non-généralisés

Premature optimization is the root of all evil.

Donald Knuth

Sommaire

Ce chapitre introduit les principales avancées et optimisations sur les tests de vacuité pouvant être utilisés dans l'approche automate pour le model checking*. Ces tests peuvent être simplifiés en fonction de l'automate à vérifier, et nous montrons dans ce chapitre qu'ils peuvent tous être exprimés dans un cadre unifié qui sera utilisé tout au long de ce manuscrit.*

3.1 Généalogie et comparaison des approches

Les tests de vacuité permettent de savoir si un automate de Büchi est vide. Dans ce manuscrit, nous nous intéressons au cas où cet automate résulte du produit synchronisé entre l'automate de la propriété ($\mathcal{A}_{\neg\varphi}$) et celui du système ($\mathcal{A}_{\mathcal{K}}$) : cet automate est noté $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$.

Nous avons vu dans le chapitre précédent qu'il existe deux grandes catégories de tests de vacuité. Ceux basés sur l'énumération des composantes fortement connexes gèrent mieux l'équité, mais ils sont mis de coté dès 1991 (détails figures 3.1 et 3.2) au profit d'algorithmes basés sur la recherche de cycles acceptants qui sont moins coûteux en mémoire.

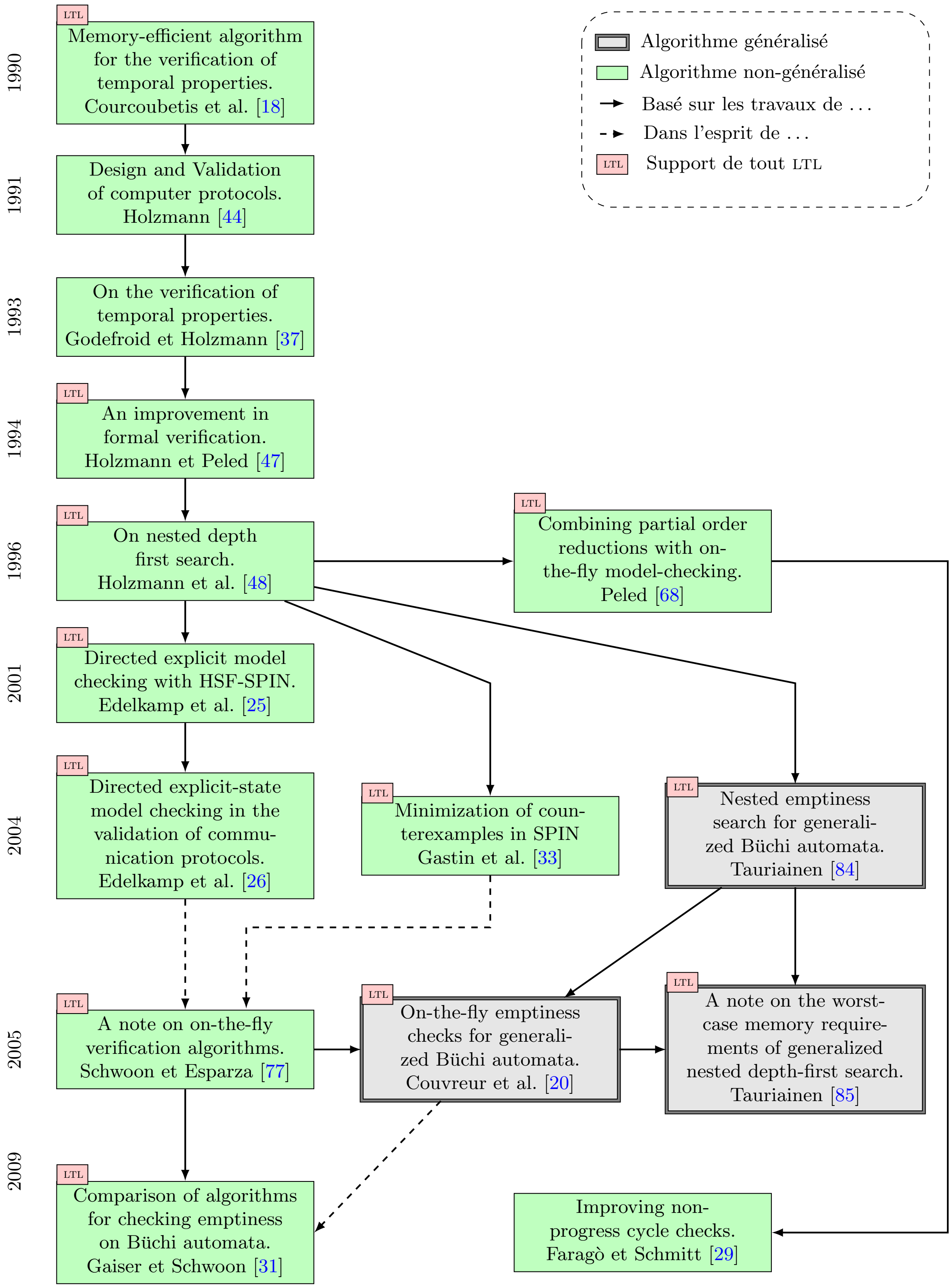


FIGURE 3.1 – Généalogie des tests de vacuité basés sur un NDFS.

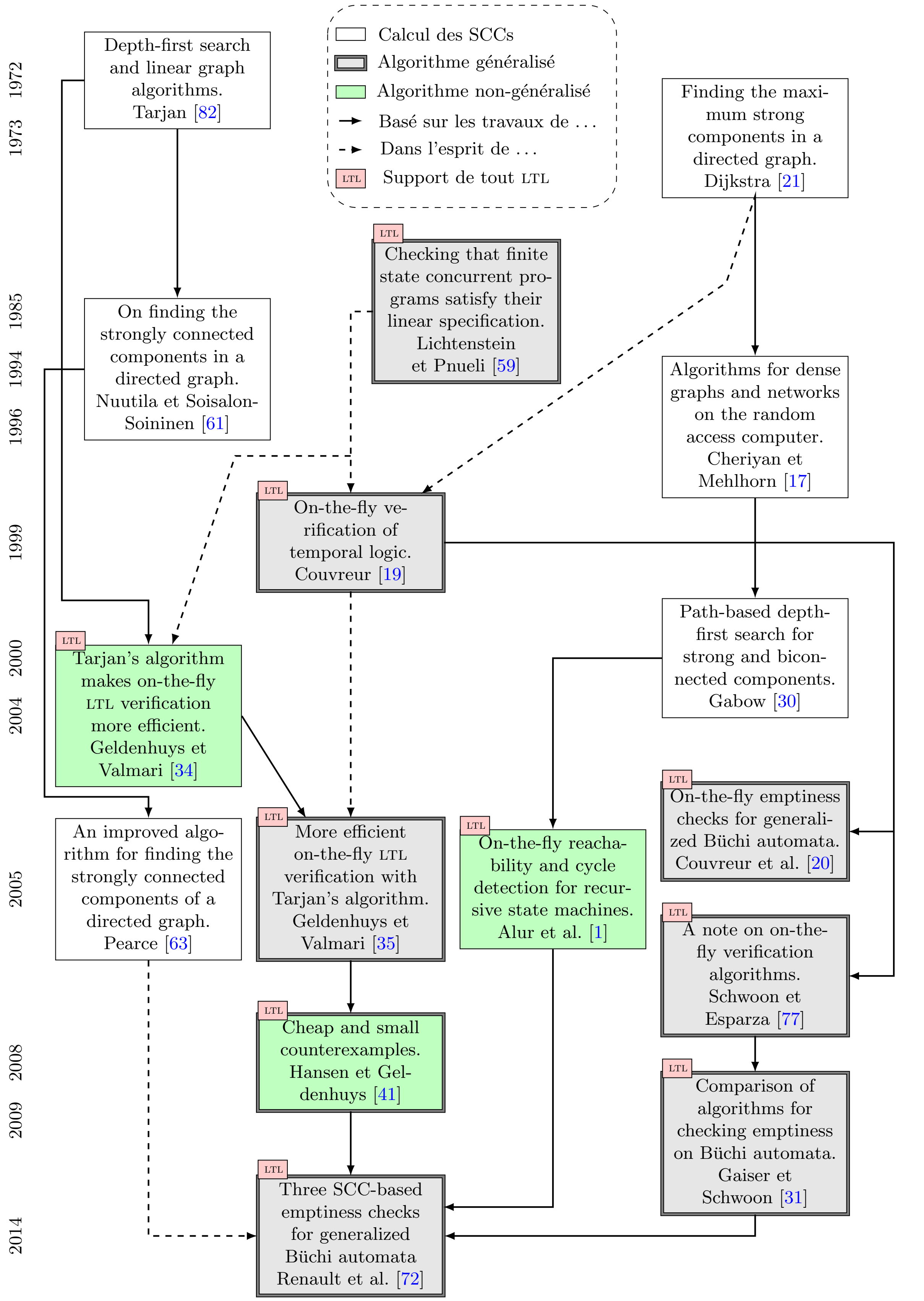


Figure 3.2 – Généalogie des tests de vacuité basés sur le calcul des composantes fortement connexes.

De nombreuses optimisations ont alors été proposées pour améliorer la vérification de propriétés de logique temporelles au moyen d'un NDFS (Nested Depth First Search). La figure 3.1 présente les principales avancées sur ces tests : les flèches ⇢ représentent les travaux partageant des idées similaires, les flèches → les travaux qui sont à la base d'autres travaux, enfin pour chaque algorithme est précisé sa capacité à gérer des automates avec plusieurs conditions d'acceptations. De manière plus générale :

- Holzmann [44] propose l'ajout d'un composant dédié au sein du modèle représentant le système pour gérer l'équité inconditionnelle ;
- Holzmann et Peled [47] proposent des techniques de réduction statiques applicables avant l'exploration par le NDFS mais ces optimisations ne sont pas compatibles avec une approche « à la volée » ;
- Godefroid et Holzmann [37] proposent un stockage hybride pour regrouper les états : ces derniers sont divisés en deux parties, la première partie servant d'index dans la table de hachage les stockant, tandis que la seconde est stockée explicitement ;
- Edelkamp et al. [26] restreignent la portée du parcours imbriqué en étudiant la structure des composantes fortement connexes de l'automate de la propriété ;
- Tauriainen [84] puis Couvreur et al. [20] s'intéressent au support d'automates généralisés pour les NDFS et montrent que leur complexité est liée au nombre de marques d'acceptation.

L'intérêt pour les tests de vacuité basés sur le calcul des composantes fortement connexes est relancé en 1999 lorsque Couvreur [19] montre qu'ils ont une complexité indépendante du nombre de marques d'acceptation, qu'ils peuvent être réalisés « à la volée » et qu'ils sont compatibles avec le *Bit State Hashing* et le *State Space Caching* (contrairement à ce qui avait été annoncé par Holzmann et Peled [47]). La figure 3.2 présente l'évolution des tests de vacuité basés sur le calcul des composantes fortement connexes : on remarque qu'ils ont effectivement été délaissés entre 1991 et 1999. Dès 2005, des travaux indépendants [20, 31, 35] convergent pour optimiser les travaux de Couvreur [19] en utilisant l'information stockée pour éviter de solliciter la fonction de transition.

Les travaux de Schwoon et Esparza [77] puis de Gaiser et Schwoon [31] clarifient les différentes avancées pour les deux familles d'algorithmes :

- lors d'une approche non-généralisée, la consommation mémoire est en faveur des algorithmes basés sur un NDFS qui ne requièrent que deux bits par états par opposition à un entier plus un bit nécessaires aux algorithmes basés sur un calcul des composantes fortement connexes ;
- les algorithmes non-généralisés basés sur un NDFS permettent une meilleure utilisation des techniques de *Bit State Hashing* et de *State Space Caching* ;
- les tests basés sur le calcul des composantes fortement connexes peuvent détecter un contre exemple dès que l'ensemble des transitions le constituant ont été visitées. Par opposition, les algorithmes basés sur un NDFS doivent attendre que tous les successeurs d'une transition acceptante aient été visités avant de lancer la procédure de détection d'un contre exemple ;
- les algorithmes basés sur un NDFS ont une complexité qui dépend du nombre de marques d'acceptation [20] là où les tests de vacuité basés sur les composantes fortement connexes ont une complexité qui en est indépendante.

La conclusion de ces travaux est sans équivoque et suggère d'utiliser des tests de vacuité basés sur le calcul des composantes fortement connexes lorsque l'automate de la propriété est fort (détails section suivante), c'est-à-dire qu'il ne possède aucunes caractéristiques permettant de simplifier le test de vacuité.

3.2 Force des automates de Büchi

La recherche d'un cycle acceptant peut être simplifiée en fonction de la *force* de l'automate de la propriété[1]. La force d'un automate est déterminée par la structure de ses composantes fortement connexes.

Définition 17 – Composante fortement connexe complète

Soit $\mathcal{A} = \langle\, Q,\, q_0,\, AP,\, \mathcal{F},\, \Delta \,\rangle$ un TGBA et $S \subseteq Q$ une composante fortement connexe. La composante fortement connexe S est dite *complète* si et seulement si :

$$\forall \ell \in 2^{AP}, \exists\, (q_1, \alpha, f, q_2) \in \Delta, \text{t.q. } q_1 \in S, q_2 \in S \text{ et } \alpha = \ell$$

Définition 18 – Force d'un automate de Büchi

La *force* d'un automate de Büchi est déduite des circuits à l'intérieur des composantes fortement connexes qui le compose. Un automate est dit :

Fort : si une composante fortement connexe peut contenir des circuits acceptants et des circuits non acceptants ;

Faible : si toutes les transitions d'une composante fortement connexe ont les même marques d'acceptation ;

Terminal : si l'automate est faible et que toutes ses composantes fortement connexes acceptantes sont complètes.

Si E_{strong} (resp. E_{weak}, resp. E_{term}) représente l'ensemble des automates forts (resp. faibles, resp. terminaux) l'inclusion suivante tient : $E_{term} \subseteq E_{weak} \subseteq E_{strong}$.

La figure 3.3 montre trois automates représentant chacun une force : on remarque que les contraintes sur les composantes fortement connexes vont par ordre croissant :

1. l'automate de gauche est composé d'une unique composante fortement connexe. C'est un automate fort car il n'y a aucune restriction sur les circuits et l'on peut avoir des circuits complètement acceptants (visitant uniquement la transition étiquetée par a) ou complètement non-acceptants (visitant uniquement la transition étiquetée par b).
2. l'automate du milieu est composé d'une unique composante fortement connexe dans laquelle l'unique circuit (visitant b puis a) n'est composé que de transitions acceptantes ;
3. enfin, l'automate de droite est composé de deux composantes fortement connexes. Celle autour de s_0 est non-acceptante et toutes les transitions ont $\emptyset$ comme marque d'acceptation. Celle autour de s_1 est acceptante puisque toutes les transitions sont étiquetées par $\bullet$; cette composante est aussi complète puisque pour chaque état de la composante il existe une transition restant dans celle-ci pour chaque lettre : cet automate est donc terminal.

1. Le test de vacuité peut aussi être simplifié lorsque l'automate à vérifier ne résulte pas d'un produit synchronisé : il suffit alors de regarder la force de cet automate.

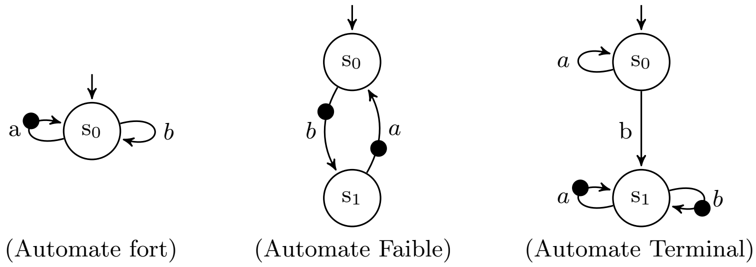


FIGURE 3.3 – Trois exemples d'automates avec des différentes forces. $AP = \{a, b\}$

Pour un automate faible ou terminal le test de vacuité peut être réduit à la recherche d'un circuit dans une composante fortement connexe acceptante. Pour les automates forts cela n'est pas suffisant, il faut s'assurer que ce circuit est aussi acceptant.

La hiérarchie proposée Manna et Pnueli [60] puis raffinée par Černá et Pelánek [15] fait le lien entre la force des automates de Büchi et certaines sous-classes de LTL. Ces dernières couvrent tout LTL et constituent une classification alternative à celle de Lamport [56] (cf. section 1.2) dans laquelle les combinaisons de propriétés de *Sûreté* et de *Vivacité* n'appartiennent à aucune classe. Cette nouvelle classification s'exprime de la manière suivante :

- *Sûreté* : exprime l'invariance d'une propriété au sein de toutes les exécutions du système ;
- *Garantie* : exprime l'assurance que quelque chose arrive au moins une fois pour toutes les exécutions du système. Ce type de propriété est généralement utilisé pour s'assurer que le système arrive dans un état symbolisant la terminaison. Dans l'exemple de la classe d'Archimède, cela correspond à un état dans lequel tous les étudiants sont dans l'état *Done* ;
- *Obligation* : exprime l'occurrence conditionnelle des propriétés de garantie et de sûreté ;
- *Récurrence* : exprime qu'une chose surviendra infiniment souvent ;
- *Persistance* : exprime qu'une propriété surviendra continuement à partir d'un certain point ;
- *Réactivité* : exprime la combinaison de toutes les propriétés précédentes et exprime l'occurrence conditionnelle des propriétés de récurrence et de persistance.

La figure 3.4 montre le lien entre force des automates, formules LTL et classes de la hiérarchie de Manna et Pnueli. Ce lien est assuré par la traduction de formules LTL en automates de Büchi proposée par Černá et Pelánek [15]. Dans cette hiérarchie, toute formule LTL est une formule de réactivité qui se traduit en un automate fort : les tests de vacuité évoqués dans la section précédente y sont donc directement applicables. Pour les formules représentant des propriétés de persistance, d'obligation, de sûreté ou de garantie, ces tests peuvent être simplifiés.

Les sections suivantes présentent les tests de vacuité pour les automates terminaux et faibles. La section 3.6 présente une adaptation du NDFS de Gaiser et Schwoon [31] pour les TBA tandis que le reste de ce manuscrit s'intéresse aux tests de vacuité supportant les automates généralisés.

3.3 DFS générique

Les tests de vacuité présentés à la section 3.1 sont tous basés sur le parcours DFS d'un TGBA. L'objectif de cette section est d'introduire un canevas de DFS générique permettant d'expri-

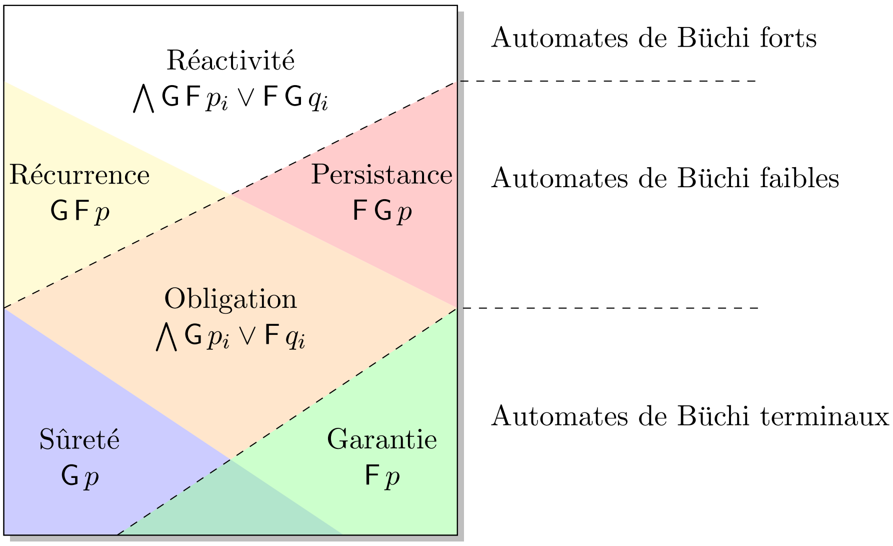


FIGURE 3.4 – Hiérarchie et classification des formules LTL avec p_i, q_i, p, et q des propositions atomiques.

mer tous les tests de vacuité mentionnés dans cette thèse : l'algorithme 1 présente cette base commune. Ce DFS utilise quatre méthodes (GET_STATUS$_{str}$, PUSH$_{str}$, POP$_{str}$, UPDATE$_{str}$), qui sont spécialisées en fonction des tests de vacuité (str) que l'on souhaite exprimer. Par la suite nous parlerons de stratégies pour désigner l'ensemble des spécialisation de méthodes nécessaires à un algorithme.

Ce DFS générique est légèrement plus compliqué qu'un DFS classique puisqu'il doit gérer les marques d'acceptation du TGBA. La variable *dfs* représente la pile du DFS et stocke des éléments de type *Step* qui doivent contenir au minimum : un état (*src*), et l'ensemble de ses successeurs (*succ*). Ces derniers sont ordonnés en respectant la politique *SuccPolicy* donnée : DEFAULT correspond à l'ordre par défaut, tandis que RANDOM est un ordre aléatoire. Lors de l'invocation, cette politique d'ordonnancement des successeurs est complétée par un paramètre *Strategy* qui permet de choisir les spécialisations à appliquer au DFS générique.

La fonction `EC`, invoquée par `sequential_main`, constitue le cœur de l'algorithme générique. Durant le parcours DFS un état peut être *vivant* (LIVE), *mort* (DEAD) ou *inconnu* (UNKNOWN). Un état est inconnu s'il n'a pas été déjà rencontré par le DFS, il est mort s'il ne peut pas faire partie d'un cycle acceptant ou aider à la détection d'un tel cycle, sinon il est vivant. Une fois un état marqué mort il ne peut plus passer à vivant ou à inconnu. De même un état vivant ne peut passer de vivant à inconnu. Le maintient de ce statut se fait alors au travers quatre méthodes (spécialisées en fonction de *str*) :

GET_STATUS$_{str}$: retourne le statut d'un état passé en paramètre ;

PUSH$_{str}$: prend en paramètre un état s et sa marque d'acceptation entrante. Au retour de cette méthode, le sommet de la pile *dfs* doit au minimum contenir s et ses successeurs, et s ne doit être inconnu ;

POP$_{str}$: dépile le sommet de la pile *dfs* ;

UPDATE$_{str}$: est appelé à chaque fois qu'une transition fermante est rencontrée. La

```
Variables Partagées :
    A : TGBA such that A = ⟨ Q, q0, AP, F, Δ ⟩        // A = A¬φ ⊗ AK
    stop : boolean                                   // Unused, see part III
Structures Globales :
    struct Transition {src : Q, acc : 2^F, dst : Q}
    struct Step       {src : Q, succ : 2^Δ}
    enum SuccPolicy {DEFAULT, RANDOM}
    enum Status     {LIVE, DEAD, UNKNOWN}
    enum Strategy   {
        Reachability,      // Section 3.4
        WeakDFS,           // Section 3.5
        NDFS,              // Section 3.6
        Tarjan,            // Section 4.1.3
        Dijkstra,          // Section 4.2.3
        Tarjan-uf,         // Section 5.2.1
        Dijkstra-uf,       // Section 5.2.2
        CNDFS,             // Section 8.3.2
        TarjanPar,         // Section 9.2
        DijkstraPar,       // Section 9.3
    }
Variables Locales :
    dfs : stack of ⟨ Step ⟩

sequential_main(str : Strategy, pol : SuccPolicy)
    stop ← ⊥
    EC(str, pol)

EC(str : Strategy, pol : SuccPolicy)
    PUSH_str(∅, q0)
    while ¬ dfs.empty() ∧ ¬ stop do
        Step step ← dfs.top()
        if step.succ ≠ ∅ then
            Transition t ← pick one from step.succ according to pol
            switch GET_STATUS_str(t.dst) do
                case DEAD
                    skip
                case LIVE
                    UPDATE_str(t.acc, t.dst)
                case UNKNOWN
                    PUSH_str(t.acc, t.dst)
        else
            POP_str(step)
    stop ← ⊤                     // Unused, see Chapter 5
```

Algorithme 1: DFS générique

marque d'acceptation et l'état destination lui sont passés en paramètre, l'état source est l'état au sommet de la pile *dfs*.

Note : L'algorithme générique utilise aussi une variable *stop* qui ne sera utilisée que pour les tests de vacuité parallèles (partie III) et qui peut donc être ignorée pour tous les algorithmes séquentiels. Nous distinguons aussi deux types de variables : les variables partagées et le variables locales. Pour tous les tests de vacuité de cette partie une telle distinction n'est pas nécessaire. En revanche pour les tests de vacuité de la partie III, cette distinction permet de savoir quelles sont les données accédées concurremment.

3.4 L'accessibilité

Lorsque $\neg\varphi$ est une formule de garantie, l'automate $\mathcal{A}_{\neg\varphi}$ est terminal (cf figure 3.4) et toutes ses composantes fortement connexes acceptantes sont complètes. Comme nous ne considérons ici que les systèmes non-bloquants, l'automate du produit $(\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}})$ est aussi terminal.

Ainsi dès que la projection d'un état accessible du produit sur $\mathcal{A}_{\neg\varphi}$ appartient à une composante acceptante de $\mathcal{A}_{\neg\varphi}$ on a l'assurance qu'un contre exemple existe. Comme cette composante est forcément complète elle va accepter toutes les exécutions du système qui passent par cet état. Notons qu'il s'agit de l'unique cas où un contre exemple peut être détecté en ne calculant que le préfixe menant au cycle acceptant.

Dans le cas où $\mathcal{A}_{\neg\varphi}$ est terminal et $\mathcal{A}_{\mathcal{K}}$ non bloquant, le test de vacuité de $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$ peut donc être réduit à un simple problème d'accessibilité. La stratégie 1 présente la spécialisation des procédures génériques à appliquer au DFS générique pour réaliser ce test.

Cette spécialisation utilise un ensemble *openset* qui stocke les états qui ont déjà été traités. A chaque fois qu'un nouvel état est découvert, la méthode $\texttt{PUSH}_{Reachability}$ utilise la fonction

```
Variables Locales supplémentaires :
  openset : hashset of ⟨ Q ⟩

PUSH_Reachability(acc ∈ 2^F, q ∈ Q) → int
  if inAcceptingSCCof(P_|A¬φ(q), A¬φ) then
    report Accepting cycle detected!
  openset.insert(⟨ q ⟩)
  dfs.push(⟨ q, succ(q) ⟩)
  return 0

GET_STATUS_Reachability(q ∈ Q) → Status
  if openset.contains(q) then
    return DEAD                // Already visited
  return UNKNOWN

UPDATE_Reachability(acc ∈ 2^F, dst ∈ Q)
  return                       // Counterexample detected before closing edge!

POP_Reachability(s ∈ Step)
  dfs.pop()
```

Stratégie 1: Accessibilité (hypothèse faite que le système n'a pas d'états bloquants).

`inAcceptingSCCof` pour vérifier si cet état appartient à une composante fortement connexe acceptante de $\mathcal{A}_{\neg\varphi}$. Si c'est le cas un contre exemple existe et il peut être retourné, sinon cet état est inséré dans la pile *dfs* pour que ses successeurs soient traités ultérieurement. Cet état est aussi inséré dans *openset* pour éviter d'avoir à le revisiter. Ainsi tous les états appartenant à l'*openset* peuvent être considérés comme morts (ils ne font partie d'aucun cycle acceptant) et la méthode $\texttt{GET\_STATUS}_{Reachability}$ peut déduire le statut d'un état par un simple test d'appartenance à *openset*. La méthode $\texttt{POP}_{Reachability}$ ne fait que de dépiler l'état au sommet de la pile *dfs* tandis que la méthode $\texttt{UPDATE}_{Reachability}$ ne fait rien puisque la détection d'un cycle acceptant sera faite avant la détection d'une transition fermante.

Cet algorithme fonctionne « à la volée » puisque les états sont découvert au fur et à mesure pour être ensuite stockés dans la variable *openset*. Celle-ci peut également être couplée à la technique du *Bit State Hashing* : si deux état q_1 et q_2 sont en collision et que q_1 a été inséré dans la variable *openset* avant q_2 alors la méthode $\texttt{GET\_STATUS}_{Reachability}$ considère que q_2 est mort. Dans ce cas, q_2 et ses successeurs sont ignorés par l'algorithme. Le *State Space Caching* peut lui aussi aisément être appliqué en bornant la taille de *openset* : lors d'un $\texttt{PUSH}_{Reachability}$ si la taille maximale est atteinte, le plus ancien état stocké est remplacé. Notons néanmoins que cette technique ne peut être appliquée sur les états de la pile du DFS pour que la terminaison du parcours soit assurée. Enfin, savoir si $\mathcal{A}_{\neg\,\varphi}$ est terminal implique seulement de savoir reconnaître syntaxiquement si $\neg\,\varphi$ est une formule de garantie. En effet, les algorithmes de traduction LTL proposés par Černá et Pelánek [15] ou par Schneider [76] assurent la production d'un automate terminal.

Note : Une exploration DFS de l'automate a ici été choisie pour une meilleure intégration avec l'algorithme générique proposé à la section précédente. Néanmoins, ce problème d'accessibilité peut être traité aussi efficacement avec un parcours BFS puisqu'il n'est pas nécessaire de détecter les arcs fermants. Néanmoins, ces parcours sont réputés pour être plus consommateurs de mémoire ce qui justifie notre choix de présentation.

3.5 Le DFS – test de vacuité faible

Lorsque $\neg\varphi$ est une formule de persistance, $\mathcal{A}_{\neg\varphi}$ peut être réalisé par un automate faible et tous les circuits des composantes fortement connexes acceptantes sont acceptants. Contrairement au cas des automates terminaux, atteindre un état du produit dont la projection sur $\mathcal{A}_{\neg\varphi}$ appartient à une composante fortement connexe acceptante n'est pas suffisant pour détecter un contre exemple : rien n'assure qu'un circuit sera formé puisque la composante n'est pas nécessairement complète.

Exemple.
La figure 3.5 présente un automate de la propriété ($\mathcal{A}_{\neg\varphi}$) qui est faible puisqu'il est composé d'une unique composante fortement connexe acceptante dont tous les chemins sont acceptants. Lors de la synchronisation avec l'automate de la structure de Kripke ($\mathcal{A}_{\mathcal{K}}$) la projection de l'état du produit s_2,q_2 sur $\mathcal{A}_{\neg\varphi}$ est dans une composante acceptante mais il n'existe pas de circuit acceptant autour de cet état dans $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$.

Comme $\mathcal{A}_{\neg\varphi}$ est faible, toutes les transitions fermantes ferment soit des circuits acceptants soit des circuits non acceptants. Ainsi détecter une transition fermante dont la destination (ou

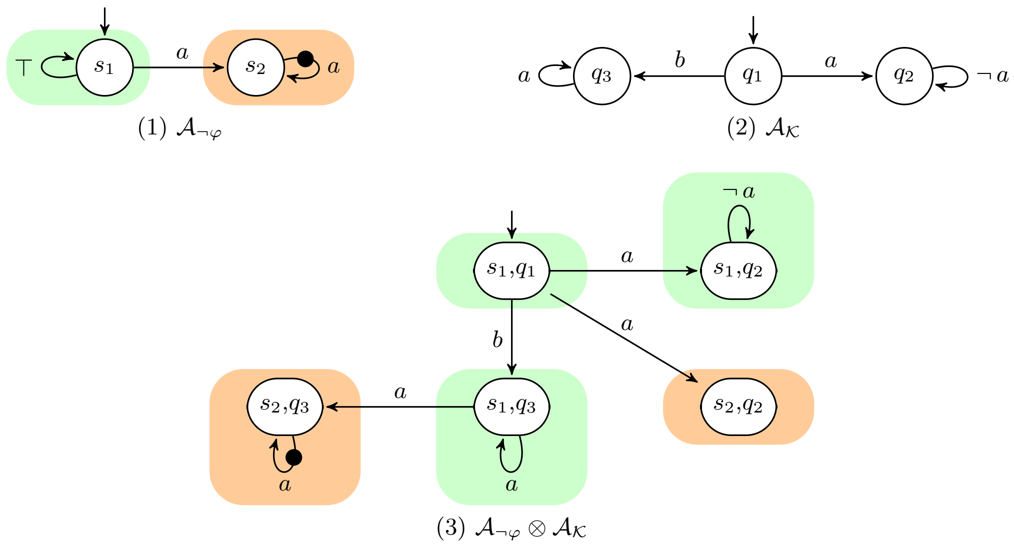


FIGURE 3.5 – $AP = \{a, b\}$. (1) Automate pour $\neg\varphi = \mathsf{F}\,\mathsf{G}\,a$. (2) Automate de la structure de Kripke du modèle. (3) Produit synchronisé.

Structures supplémentaires :
  enum *wcolor* { *OnDFS*, *Visited* }

Variables Locales supplémentaires :
  coloredset : *map* of $Q \mapsto \langle c : wcolor \rangle$

`PUSH`$_{WeakDFS}$`(`$acc \in 2^{\mathcal{F}}$, $q \in Q$`)` $\rightarrow$ *int*
  coloredset.`insert(`$\langle$ q, *OnDFS* $\rangle$`)`
  dfs.`push(`$\langle$ q, `succ(`q`)` $\rangle$`)`

`GET_STATUS`$_{WeakDFS}$`(`$q \in Q$`)` $\rightarrow$ *Status*
  if *coloredset*.`contains(`q`)` **then**
    return *coloredset*.`get(`q`)`.c = *Visited* ? DEAD : LIVE
  return UNKNOWN

`UPDATE`$_{WeakDFS}$`(`$acc \in 2^{\mathcal{F}}$, $dst \in Q$`)`
  if `inAcceptingSCCof(`$\mathcal{P}_{|\mathcal{A}_{\neg\varphi}}(dst)$, $\mathcal{A}_{\neg\varphi}$`)` **then**
    report Accepting cycle detected !
  return

`POP`$_{WeakDFS}$`(`$s \in Step$`)`
  coloredset.`get(`$s.src$`)`.$c \leftarrow$ *Visited*
  dfs.`pop()`

Stratégie 2: Test de vacuité pour les produits avec un automate de la propriété faible.

Variables Locales supplémentaires :
  ondfsset : *set* of $\langle Q \rangle$
  visitedset : *set* of $\langle Q \rangle$

`PUSH`$_{WeakDFS}$`(`$acc \in 2^{\mathcal{F}}$, $q \in Q$`)` $\rightarrow$ *int*
  ondfsset.`insert(`$\langle$ q, *OnDFS* $\rangle$`)`
  dfs.`push(`$\langle$ q, `succ(`q`)` $\rangle$`)`

`GET_STATUS`$_{WeakDFS}$`(`$q \in Q$`)` $\rightarrow$ *Status*
  if *ondfsset*.`contains(`q`)` **then**
    return LIVE ;
  else if *visitedset*.`contains(`q`)` **then**
    return DEAD ;
  return UNKNOWN

`UPDATE`$_{WeakDFS}$`(`$acc \in 2^{\mathcal{F}}$, $dst \in Q$`)`
  if `inAcceptingSCCof(`$\mathcal{P}_{|\mathcal{A}_{\neg\varphi}}(dst)$, $\mathcal{A}_{\neg\varphi}$`)` **then**
    report Accepting cycle detected !
  return

`POP`$_{WeakDFS}$`(`$s \in Step$`)`
  visitedset.`insert(`$s.src$`)`
  ondfsset.`remove(`$s.src$`)`
  dfs.`pop()`

Stratégie 3: Test de vacuité pour les produits avec un automate de la propriété faible (compatible Bit State Hashing et State Space Caching).

la source) est dans une composante fortement connexe acceptante suffit à trouver un contre exemple. Pour les automates faibles, le test de vacuité peut donc être réduit à la recherche de transitions fermantes retournant sur la pile du DFS. Ce test est présenté stratégie 2 et chaque état est alors associé à une couleur indiquant s'il est sur la pile *dfs* (*OnDFS*) ou non (*Visited*). Les nouveaux états sont insérés dans *coloredset* avec la couleur *OnDFS* lors d'un $\texttt{PUSH}_{WeakDFS}$. Lors d'un $\texttt{POP}_{WeakDFS}$ ces états sont simplement marqués comme étant visités. Dès qu'une transition fermante est détectée lors d'un $\texttt{UPDATE}_{WeakDFS}$, il suffit de vérifier si la projection de la destination sur $\mathcal{A}_{\neg\varphi}$ appartient à une composante acceptante.

Ce test de vacuité fonctionne lui aussi « à la volée » car détecter que deux états sont dans une même composante fortement connexe acceptante de $\mathcal{A}_{\neg\varphi}$ peut être fait en regardant si $acc = 2^{\mathcal{F}}$ (dans le $\texttt{UPDATE}_{WeakDFS}$) puisque les transitions des composantes acceptantes des automates faibles sont étiquetées par $\mathcal{F}$[2]. L'algorithme est aussi compatible avec le *State Space Caching* et le *Bit State Hashing* à une unique restriction : les états sur la pile *dfs* ne doivent pas être la cible de ces techniques. Ces techniques ne peuvent donc être appliquées que sur les états marqués *Visited* : la stratégie 3 propose une adaptation le permettant. Les seules modifications sont : l'ajout des structures *visitedset* (ligne 3) et *ondfsset* (ligne 2) stockant respectivement les états morts et ceux sur la pile *dfs*. La couleur d'un état est alors implicite et seules les méthodes $\texttt{GET\_STATUS}_{WeakDFS}$ et $\texttt{POP}_{WeakDFS}$ sont alors impactées par ces changements.

Note : Dans le cas des formules de garantie et lorsque le système n'est pas sans blocage ou qu'on ne peut le savoir à l'avance, c'est ce test de vacuité qui doit être appliqué.

3.6 Le NDFS optimisé

Les tests de vacuité basés sur les NDFS ont été massivement étudiés sur les automates de Büchi (BA). Dans cette section, nous présentons un test de vacuité qui travaille directement sur les TBA et qui intègre les principales techniques d'optimisation existantes.

3.6.1 Détails de l'algorithme

Si l'automate de la propriété n'est ni terminal ni faible alors il est fort et il possède une composante fortement connexe qui peut contenir aussi bien des circuits acceptants que des circuits non-acceptants. Avec ce type d'automate, savoir si la cible d'une transition fermante est dans une composante acceptante de $\mathcal{A}_{\neg\varphi}$ ne suffit plus à détecter un contre exemple : il est nécessaire d'analyser le circuit pour savoir s'il est acceptant.

2. Pour des raisons de compatibilité avec les techniques présentés au chapitre 7 nous ne testons pas directement $acc = \mathcal{F}$ à la ligne 13

```
Structures supplémentaires :
  enum color { Blue, Red, Cyan }
  struct Step {src : Q,    succ : 2^Δ,
               acc : 2^F,  allred : bool} // Refinement of Step of Algo. 1

Variables Locales supplémentaires :
  coloredset : map of Q ↦ ⟨c : color⟩

PUSH_NDFS(acc ∈ 2^F, q ∈ Q) → int
    coloredset.insert(⟨ q, Cyan ⟩)
    dfs.push(⟨ q, succ(q), acc, ⊤ ⟩)

GET_STATUS_NDFS(q ∈ Q) → Status
    if coloredset.contains(q) then
        return coloredset.get(q).c = Red ? DEAD : LIVE
    return UNKNOWN

UPDATE_NDFS(acc ∈ 2^F, dst ∈ Q)
    if acc = 2^F then
        if coloredset.get(dst).c = Cyan then
            report Accepting cycle detected!
        else
            nested_dfs(dst)
            coloredset.get(dst).c ← Red
    else
        dfs.top().allred ← ⊥

POP_NDFS(s ∈ Step)
    dfs.pop()
    coloredset.get(s.src).c ← Blue
    if s.allred = ⊤ then
        coloredset.get(s.src).c ← Red
    else if s.acc = 2^F then
        nested_dfs(s.src)
        coloredset.get(s.src).c ← Red
    else
        if ¬ dfs.empty() then
            dfs.top().allred ← ⊥

// Also known as Red-DFS
nested_dfs(q ∈ Q)
    foreach Transition t ∈ succ(q) do
        if coloredset.get(t.dst).c = Cyan then
            report Accepting cycle detected!
        if coloredset.get(t.dst).c = Blue then
            coloredset.get(t.dst).c ← Red
            nested_dfs(t.dst)
```

Stratégie 4: NDFS tel que présenté par Gaiser et Schwoon [31]

Note : Contrairement aux deux tests de vacuité présentés précédemment, ce test (et tous ceux qui seront présentés par la suite à l'exception de ceux du chapitre 7) fonctionne sur des automates qui ne résultent pas nécessairement d'un produit entre l'automate de la propriété et celui du modèle.

Les tests de vacuité basés sur un NDFS utilisent un premier parcours (traditionnellement appelé bleu) qui va explorer l'automate. Dès qu'une transition acceptante allant vers un état vivant est détectée, un second parcours (traditionnellement appelé rouge) est lancé. Ce second parcours cherche à atteindre l'état source de la transitions acceptante pour exhiber un cycle acceptant.

La stratégie 4 présente cet algorithme dans le cadre du DFS générique avec les optimisations suggérées par Gaiser et Schwoon [31]. À tout moment un état est *Cyan* s'il est sur la pile *dfs*, *Red* s'il a été détecté comme ne faisant pas parti d'un cycle acceptant, *Blue* s'il a été exploré par le premier parcours mais n'est plus sur la pile *dfs*. Un état ne peut donc passer que de *Cyan* à *Blue*, puis de *Blue* à *Red*. La méthode `nested_dfs`, déclenchée lors d'un $\texttt{POP}_{\text{NDFS}}$ ou lors d'un $\texttt{UPDATE}_{\text{NDFS}}$, permet alors de lancer ce parcours imbriqué et de marquer les états comme rouges : si un état sur la pile *dfs* est rencontré par ce second parcours un contre exemple est détecté puisqu'on a l'assurance d'atteindre l'état ayant déclenché ce parcours, sinon tous les états visités sont marqués rouges.

Cet algorithme raffine la structure *Step* pour stocker deux informations supplémentaires : la condition portée sur la transition entrante d'un état (*acc*) et la variable *allred* qui permet de savoir si l'on peut marquer un état comme étant rouge prématurément. Cette dernière optimisation (proposée par Gaiser et Schwoon [31]) est une adaptation de celle de Gastin et al. [33] qui permet de réduire la portée des parcours imbriqués en marquant des états (qui ne peuvent atteindre des états vivants) comme rouges dans le premier parcours DFS La méthode $\texttt{PUSH}_{\text{NDFS}}$ est alors modifiée pour prendre en compte ces changements : les nouveaux états sont marqués *Cyan* et la variable *allred* est positionnée à $\top$.

Comparé aux stratégies précédentes, cet algorithme est plus coûteux en mémoire puisque deux bits par états sont nécessaires pour stocker *acc* et *allred* ; mais aussi plus coûteux en temps puisque les états peuvent être visités deux fois. Cet algorithme fonctionne lui aussi « à la volée » et les modifications à apporter pour le rendre compatible avec le *Bit State Hashing* et le *State Space Caching* sont proches des modifications proposées pour le test de vacuité d'automates faibles : les état *Cyan* doivent être stockés sans erreur tandis que les autres peuvent être oubliés ou souffrir des collisions.

Note : Cet algorithme requiert que les états soient visités dans le parcours rouge et dans le parcours bleu dans le même ordre pour être valide. Ainsi la politique RANDOM (page 41) ne peut être utilisée que si chaque successeur de chaque état est tiré dans le même ordre (pseudo-aléatoire).

3.6.2 Déroulement de l'algorithme

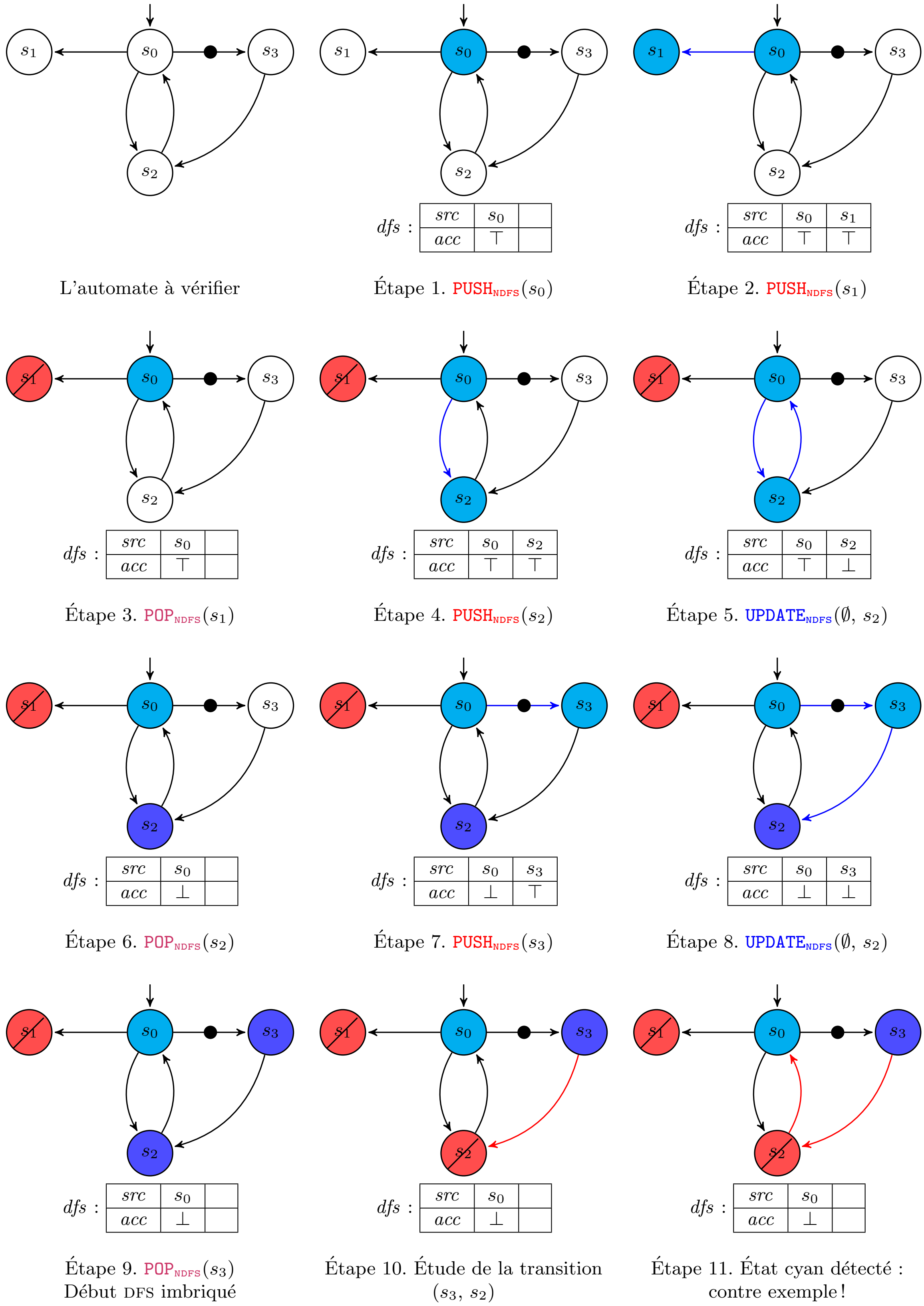


FIGURE 3.6 – Déroulement du NDFS

Exemple.
La figure 3.6 présente le déroulement de cet algorithme sur un exemple simple. Les étiquettes sur les transitions sont omises pour plus de clarté et, comme ce test de vacuité fonctionne sur n'importe quel type d'automate non généralisé, il n'est nécessaire de connaître ni l'automate de la structure de Kripke ni celui de la propriété. De plus pour chaque étape est présenté un extrait utile de la pile *dfs*.

Les étapes 1 et 2	: montrent la progression du parcours DFS : les états s_0 et s_1 sont découverts et marqués cyan ;
L'étape 3	: détaille le marquage de l'état s_1 en rouge : comme il n'a pas de successeurs, la variable *allred* associée dans *dfs* est à $\top$ et il peut directement être marqué rouge ;
L'étape 4	: montre la reprise du parcours DFS, et la coloration de l'état s_2 en cyan ;
L'étape 5	: montre la détection de la transition fermante (s_2, s_0). Comme elle n'est pas acceptante le seul impact de cette opération est le passage à $\perp$ de la variable *allred* de l'élément au sommet de la pile *dfs* ;
L'étape 6	: montre simplement le marquage de l'état s_2 en bleu ;
L'étape 7	: présente la reprise du parcours DFS et la coloration de l'état s_3 en cyan ;
L'étape 8	: montre la détection de la transition (s_3, s_2). Comme cette transition n'est pas acceptante donc aucun parcours imbriqué n'est lancé et la variable *allred* au sommet de l'élément au sommet de la pile *dfs* passe à $\perp$;
L'étape 9	: décrit la suppression de l'état s_3 de la pile *dfs*. Cet état est alors marqué en bleu et comme la transition entrante sur cet état est acceptante, la procédure `nested_dfs` est lancée et la coloration des états bleus en rouge peut commencer ;
L'étape 10	: montre l'évolution du parcours imbriqué et la découverte de la transition (s_3, s_2). L'état s_2 est alors marqué comme rouge ;
L'étape 11	: détaille la découverte de l'arc (s_2, s_0) par le parcours rouge. Lors de cette détection, un état cyan (s_0) est découvert : un contre exemple est alors détecté. Ce contre exemple est extrait en suivant les piles des parcours bleus et rouges. Le cycle $(s_0, s_3)(s_3, s_1)(s_1, s_0)$ constitue donc l'exécution acceptante et ne possède pas de préfixe puisque le dernier état du cycle est l'état initial.

3.7 Conclusion

Généralement les tests de vacuité sont présentés avec des structures de contrôle qui leur sont propres, ce qui les rend difficilement comparables. Dans ce chapitre, nous avons fourni un effort d'homogénéisation et de factorisation qui permet d'avoir une vue homogène de ces algorithmes. Le canevas générique présenté dans ce chapitre sera ensuite utilisé pour exprimer tous les tests de vacuité mentionnés dans cette thèse.

De plus, nous avons vu que les tests de vacuité peuvent être raffinés en fonction de l'automate de la propriété. Ces tests traitent efficacement les automates de Büchi ayant qu'une seule condition d'acceptation mais ils ne sont pas les seuls. Les algorithmes de Geldenhuys et Valmari [34] ou Alur et al. [1] le permettent aussi mais sont basés sur le calcul des composantes fortement connexes ce qui les rend plus facilement généralisable [35].

L'inconvénient des algorithmes basés sur la recherche des cycles acceptant, lorsque l'on s'intéresse à des automates généralisés, est qu'ils nécessitent le déclenchement d'un parcours imbriqué dès qu'une transition porte une marque d'acceptation. La complexité de ces algorithmes est alors liée au nombre de marque d'acceptations [84].

Dans le chapitre suivant nous allons voir qu'il existe cependant des algorithmes qui ont une complexité indépendantes du nombre de conditions d'acceptations utilisé par l'automate et qui semblent donc plus adaptés pour vérifier des systèmes sous hypothèses d'équités par exemple. Ces algorithmes sont tous basés sur la recherche de composantes fortement connexes et le maintient pour chacune d'entre-elles des marques d'acceptation qui y sont présentes. Le chapitre suivant montre comment ces algorithmes fonctionnent et comment ils peuvent être améliorés.

Chapitre 4

Revisiter les algorithmes de calcul de composantes fortement connexes

Simplicity is prerequisite for reliability.

Edsger W. Dijkstra

Sommaire

Le chapitre 3 a montré que le test de vacuité se réduit à un simple parcours DFS *dans le cas où l'automate de la négation de la formule est faible ou terminal. Pour les automates forts, l'algorithme généralement utilisé (*NDFS*) peut calculer deux fois les successeurs de chaque état et ne gère pas efficacement les marques d'acceptation multiples. Les tests de vacuité basés sur l'énumération des composantes fortement connexes de l'automate du produit pallient ces problèmes en ne générant qu'une fois les successeurs d'un état et en maintenant les marques d'acceptation présentes dans chaque composante.*

L'énumération des composantes fortement connexes d'un graphe a été étudiée indépendemment par Tarjan [82] et Dijkstra [21] au début des années 1970. Ces deux algorithmes ont la même complexité en temps et en mémoire dans le pire cas, mais des subtilités, pouvant impacter les tests de vacuité basés sur ces algorithmes, existent.

4.1 Test de vacuité basé sur l'algorithme de Tarjan

Dans cette section, nous nous intéressons d'abord à l'algorithme de calcul des composantes fortement connexes d'un graphe proposé par Tarjan, puis nous montrons comment le modifier pour le transformer en un test de vacuité pour automates de Büchi généralisés.

4.1.1 Le calcul des composantes fortement connexes

L'algorithme proposé par Tarjan est probablement le plus connu des algorithmes d'énumération des composantes fortement connexes. Il cherche à détecter la *racine* de chaque composante, c'est-à-dire le premier état de celle-ci rencontré lors du DFS. Lors du parcours, tous les états vivants (LIVE) sont associés à un identifiant appelé LIVE *number*. Cet identifiant est tel que pour deux états vivants s_1 et s_2, le LIVE *number* de s_1 doit être petit que celui de s_2 si et seulement si s_1 a été découvert avant s_2 par le DFS. La racine d'une composante fortement connexe est donc l'état de la composante qui a le plus petit LIVE *number*.

L'algorithme stocke alors, pour chaque état de la pile DFS, le plus petit LIVE *number* accessible depuis cet état au sein d'une variable traditionnellement appelée *lowlink*. Cette variable est mise à jour à chaque fois qu'une transition fermante est détectée et à chaque fois qu'un état est dépilé de la pile DFS. Lorsque cet état est dépilé, si son *lowlink* est égal à son LIVE *number* alors cet état est une racine. Dès qu'une racine est détectée, tous les états de la même composante peuvent être marqués comme morts (DEAD) et il n'est plus utile de maintenir leur LIVE *number*. Les états qui appartiennent à la même composante fortement connexe sont les états qui on un LIVE *number* plus grand que la racine.

La stratégie 5 présente cet algorithme dans le cadre du DFS générique présenté au chapitre précédent. Les encadrés (bleus) peuvent être ignorés pour le moment (ils seront discutés section 4.1.3). Quatre variables globales sont nécessaires :

- *visitedset* : permet de stocker les états dont tous les successeurs ont été visités ;
- *live* : stocke tous les états vivants qui ne sont plus sur la pile DFS ;
- *livenum* : associe chaque état à son LIVE *number* ;
- *llstack* : stocke pour chaque état de la pile *dfs* le *lowlink* associé.

La pile *llstack* (de type *pstack* – elle stocke des positions) stocke des entiers et se manipule au travers des méthodes : `pop` qui supprime l'élément au sommet de la pile, `top` qui retourne l'élément au sommet de la pile, `pushnontransient` et `pushtransient` qui ajoutent un élément au sommet de la pile. L'utilisation d'une telle pile permet des optimisations qui seront détaillées en section 4.4. Pour le moment, les paramètres des méthodes `pop` et `top` peuvent être ignorés. De la même manière, les méthodes `pushnontransient` et `pushtransient` peuvent être vues comme une seule et unique méthode qui ajoute un élément au sommet de la pile.

La méthode $\texttt{PUSH}_{Tarjan}$ marque chaque nouvel état comme vivant et lui associe un identifiant dans *livenum* : il s'agit de son LIVE *number* (ligne 11). Cet identifiant est alors inséré dans les piles *dfs* et *llstack* et deux états vivants ne peuvent pas avoir le même LIVE *number* puisque leur attribution est faite selon la taille de *livenum*. Un état (non racine) n'est inséré dans la pile *live* qu'au moment de sa suppression de *dfs* lors d'un $\texttt{POP}_{Tarjan}$: il s'agit de l'optimisation de Nuutila

```
Structures supplémentaires :
  struct Step {src : Q,    succ : 2^Δ,
               acc : 2^F,  pos : int} // Refinement of Step of Algo. 1

Variables Locales supplémentaires :
  live : stack of ⟨ Q ⟩
  livenum : map of Q ↦ ⟨p : int⟩
  visitedset : stack of ⟨ Q ⟩
  llstack : pstack of ⟨ p : int, acc : 2^F ⟩

PUSH_Tarjan(acc ∈ 2^F, q ∈ Q) → int
  p ← livenum.size()
  livenum.insert(⟨ q, p ⟩)
  llstack.pushtransient(p)
  dfs.push(⟨ q, succ(q), acc, p ⟩)

GET_STATUS_Tarjan(q ∈ Q) → Status
  if livenum.contains(q) then
    return LIVE
  if visitedset.contains(q) then
    return DEAD
  return UNKNOWN

UPDATE_Tarjan(acc ∈ 2^F, dst ∈ Q)
  ⟨ p, a ⟩ ← llstack.pop(dfs.top().pos)
  llstack.pushnontransient( min(p, livenum.get(d)), acc ∪ a )
  if acc ∪ a = F then
    report Accepting cycle detected!

POP_Tarjan(s ∈ Step)
  dfs.pop()
  ⟨ ll, acc ⟩ ← llstack.pop(s.pos)
  // s.src is a root.
  if ll = s.pos then
    // Mark this SCC as Dead.
    livenum.remove(s.src)
    visitedset.insert(s.src)
    while livenum.size() > s.pos do
      q ← live.pop()
      livenum.remove(q)
      visitedset.insert(q)
  else
    // s.src is not a root.
    ⟨ p, a ⟩ ← llstack.pop(dfs.top().pos)
    llstack.pushnontransient( min(p, ll), a ∪ acc ∪ s.acc )
    if a ∪ acc ∪ s.acc= F then
      report Accepting cycle detected!
    live.push(s.src)
```

Stratégie 5: Test de vacuité basé sur l'algorithme de Tarjan .

et Soisalon-Soininen [61] [1]. Lors d'une telle opération, le *lowlink* (stocké dans *llstack*) de l'état qui est en train d'être dépilé et celui de son prédécesseur sont comparés pour ne conserver que le plus petit, i.e. seul le plus petit *live number* accessible depuis un état est conservé. Une telle mise à jour est aussi effectuée à chaque fois qu'une transition fermante est détectée lors d'un $\texttt{UPDATE}_{Tarjan}$. Une racine est alors détectée lors d'un $\texttt{POP}_{Tarjan}$ en regardant si son *lowlink* est resté inchangé durant le parcours : dans ce cas tous les états de la même composante fortement connexe sont transférés de *live* vers *visitedset* et les LIVE *numbers* associés dans *livenum* peuvent être réutilisés. Enfin, la méthode $\texttt{GET\_STATUS}_{Tarjan}$ se charge seulement de regarder la présence d'un état dans *visitedset* pour le déclarer DEAD, ou son absence dans *livenum* pour le déclarer UNKNOWN. Tous les autres états doivent donc être considérés comme LIVE.

4.1.2 Déroulement de l'algorithme

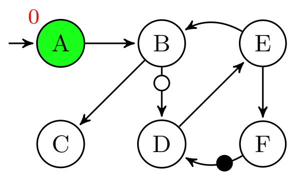
Exemple.
La figure 4.1 présente l'évolution de l'algorithme de Tarjan sur un exemple : les états blancs sont les états non-encore visités, les verts ceux déjà visités et les gris barrés ceux qui sont morts. Les numéros rouges représentent les LIVE *numbers* associés à chaque état. Pour chaque étape est représenté la pile *llstack* qui évolue parallèlement à *dfs*. Les marques d'acceptation de la pile *llstack* et de l'automate peuvent être ignorées pour le moment puisqu'elles sont inutiles pour le calcul des composantes fortement connexes.

Les étapes 1 à 3 présentent la découverte des états A, B, C : les LIVE *numbers* associés sont insérés dans *llstack*. Comme l'état C ne possède aucun successeur, il peut être marqué comme mort à l'étape 4 : il est transféré de *livenum* vers *visitedset*, son *lowlink* est supprimé de *llstack*, et son LIVE *number* supprimé de *dfs*. Ce dernier est ensuite recyclé à l'étape 5 lors de la découverte l'état D. L'étape 7 montre la découverte de la transition fermante (E, B) : le *lowlink* associé à E référence alors le LIVE *number* de l'état B. De la même manière l'étape 9 montre la découverte de la transition fermante (F, D) et la mise à jour du *lowlink* associé à F. L'étape 13 montre la détection de la racine de la composante fortement connexe formée par les états B, D, E et F. Tous ces états sont alors transférés vers *visitedset*.

Notons que l'optimisation suggérée par Nuutila et Soisalon-Soininen [61] (et intégrée ici) évite de rajouter B (et de manière plus générale toutes les racines) dans *live*. L'énumération des composantes fortement connexes se fait donc pendant un $\texttt{POP}_{Tarjan}$ lorsque la racine est détectée. L'algorithme présenté ici détecte donc trois composantes pour l'automate donné en exemple.

L'approche proposée par Tarjan utilise donc les transitions fermantes pour accumuler l'information d'appartenance à une même composante fortement connexe. Cette information est remontée lorsque les états sont dépilés de *dfs*. Dans le cadre d'un test de vacuité, cette approche est intéressante car à chaque transition visitée une information d'appartenance à une composante fortement connexe est stockée. Il est assez naturel de combiner cette information avec une indication des marques d'acceptation rencontrées.

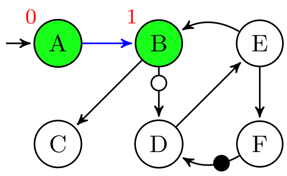
1. Dans la pratique cette optimisation est très efficaces lorsque l'automate a beaucoup de composantes fortement connexes, cf. chapitre 6.



llstack:

0				
∅				

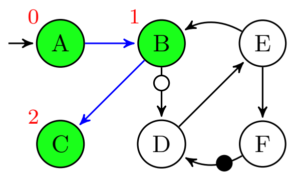
Étape 1. $\mathtt{PUSH}_{Tarjan}(\emptyset,\ A)$



llstack:

0	1			
∅	∅			

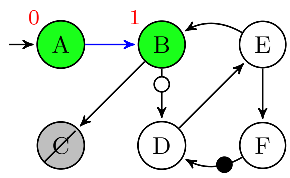
Étape 2. $\mathtt{PUSH}_{Tarjan}(\emptyset,\ B)$



llstack:

0	1	2		
∅	∅	∅		

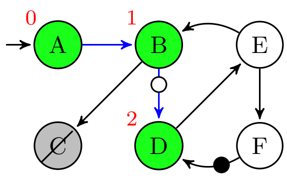
Étape 3. $\mathtt{PUSH}_{Tarjan}(\emptyset,\ C)$



llstack:

0	1			
∅	∅			

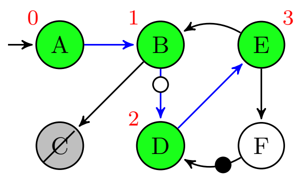
Étape 4. $\mathtt{POP}_{Tarjan}(C)$



llstack:

0	1	2		
∅	∅	∅		

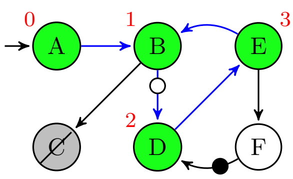
Étape 5. $\mathtt{PUSH}_{Tarjan}(\circ,\ D)$



llstack:

0	1	2	3	
∅	∅	∅	∅	

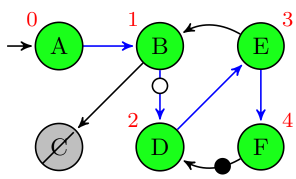
Étape 6. $\mathtt{PUSH}_{Tarjan}(\emptyset,\ E)$



llstack:

0	1	2	1	
∅	∅	∅	∅	

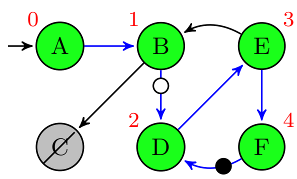
Étape 7. $\mathtt{UPDATE}_{Tarjan}(\emptyset,\ B)$



llstack:

0	1	2	1	4
∅	∅	∅	∅	∅

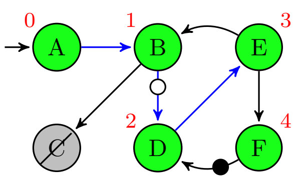
Étape 8. $\mathtt{PUSH}_{Tarjan}(\emptyset,\ F)$



llstack:

0	1	2	1	2
∅	∅	∅	∅	●

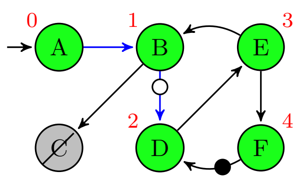
Étape 9. $\mathtt{UPDATE}_{Tarjan}(\emptyset,\ D)$



llstack:

0	1	2	1	
∅	∅	∅	●	

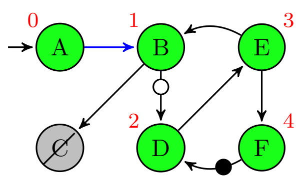
Étape 10. $\mathtt{POP}_{Tarjan}(F)$



llstack:

0	1	1		
∅	∅	●		

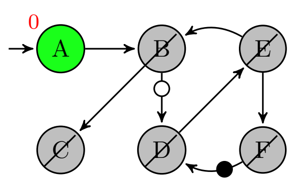
Étape 11. $\mathtt{POP}_{Tarjan}(E)$



llstack:

0	1			
∅	●○			

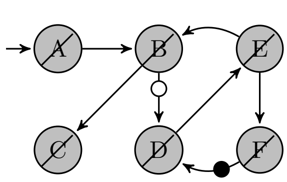
Étape 12. $\mathtt{POP}_{Tarjan}(D)$



llstack:

0				
∅				

Étape 13. $\mathtt{POP}_{Tarjan}(B)$



llstack:

Étape 14. $\mathtt{POP}_{Tarjan}(A)$

Figure 4.1 – Calcul des SCCs et test de vacuité basé sur l'algorithme de Tarjan

4.1.3 Le test de vacuité

À notre connaissance, l'unique test de vacuité basé strictement sur le calcul des composantes fortement connexes de Tarjan a été proposé par Geldenhuys et Valmari [34] [2]. Cet algorithme ne fonctionne que sur des automates non généralisés et conserve, pour chaque état, le LIVE *number* du dernier état acceptant sur la pile *dfs*. Comme l'automate n'a qu'une seule marque d'acceptation, un contre-exemple peut être reporté dès qu'une transition ferme un cycle contenant un état acceptant. Cette détection est faite dès toutes les transitions constituant le cycle acceptant ont été vues. Deux inconvénients subsistent néanmoins puisque ce test de vacuité :

1. stocke en plus du dernier état acceptant sur *dfs*, un *lowlink* par état vivant (pour permettre une extraction rapide des contre-exemples) là où l'algorithme de Tarjan n'en stocke qu'un par état de la pile *dfs* ;
2. n'est pas généralisé et le surcoût induit par les différentes structures par rapport au NDFS ne permet qu'une détection plus rapide des cycles acceptants [77].

Pour remédier à cela nous avons proposé [72] dans un premier temps de réduire la consommation mémoire en ne stockant des *lowlinks* que pour les états de *dfs* comme dans l'algorithme proposé originalement par Tarjan. Pour le support d'automates généralisés nous proposons d'associer une marque d'acceptation à chaque *lowlink* dans *llstack*. Lorsqu'une transition fermante est découverte, les marques qu'elle porte sont accumulées au sommet de la pile *llstack*. De la même manière, lorsqu'un état qui n'est pas une racine est dépilé de *dfs*, toutes les marques d'acceptation qui lui étaient associées sont transférées à son prédécesseur.

La stratégie 5 (en prenant en compte les encadrés bleus, page 65) présente ce test de vacuité généralisé. La pile *llstack* stocke maintenant un ensemble de marques d'acceptation en plus du *lowlink*. La méthode `pushtransient` ajoute un ensemble d'acceptation vide au sommet de la pile *llstack* tandis que la méthode `pushnontransient` permet de spécifier cet ensemble. Celui-ci sert alors d'accumulateur contenant les marques découvertes (ou remontées) jusqu'à cet état. Lors d'un $\texttt{POP}_{Tarjan}$ ou lors d'un $\texttt{UPDATE}_{Tarjan}$ il suffit de tester si toutes les marques d'acceptation ont été accumulées pour détecter l'existence d'une composante acceptante et donc d'un cycle acceptant.

Exemple.
En reprenant la figure 4.1 on note que la marque ● est progressivement remontée jusqu'à la racine (à savoir B). À l'étape 10, lorsque F est dépilé, le *lowlink* de E reste inchangé puisqu'il connaît un chemin remontant plus haut dans la pile *dfs*. En revanche, son ensemble d'acceptation est modifié pour intégrer la marque ● dont il n'avait pas connaissance. La marque ○ stockée dans la pile *dfs* n'est accumulée qu'au dépilement de l'état D à l'étape 12 : un contre-exemple est alors détecté. Et l'algorithme termine aussitôt (les étapes 13 et 14 ne sont la que pour illustrer le calcul des composantes fortement connexes).

Contrairement à l'algorithme de Geldenhuys et Valmari [34] l'algorithme proposé ici ne permet pas nécessairement la détection de contre exemple dès que toutes les transitions formant le cycle acceptant ont été visitées car l'information n'est agrégée que lorsqu'un état est dépilé

2. Certains travaux [35, 41] se revendiquent à tort de l'algorithme de Tarjan par méconnaissance ou redécouverte de l'algorithme de Dijkstra (détails section suivante).

ou lors de la détection d'une transition fermante. Cet algorithme reste néanmoins compatible avec les techniques de Bit *State Hashing* et de *State Space Caching* qui ne s'appliquent que sur l'ensemble *visitedset*. La construction de l'automate « à la volée » est quant à elle complètement applicable puisque les états sont découverts et insérés dans *livenum* dynamiquement.

4.2 Test de vacuité basé sur l'algorithme de Dijkstra

Cette section présente un autre algorithme de calcul des composantes fortement connexes, puis un test de vacuité qui est basé dessus.

4.2.1 Le calcul des composantes fortement connexes

Dans l'algorithme de Tarjan, l'information d'appartenance à une composante fortement connexe n'est transmise d'un état vers son prédécesseur qu'une fois tous les successeurs de cet état visités. L'algorithme proposé par Dijkstra [21] remonte systématiquement cette information vers la racine. Comme une racine ne peut être découverte qu'après la visite de la dernière transition de la composante, la notion de *racine potentielle* est utilisée. Une racine potentielle est un état de la pile DFS pour lequel l'algorithme ne connaît pas d'état appartenant à la même composante dont le LIVE *number* est plus petit. Les positions dans la pile *dfs* de ces racines potentielles sont stockées dans une pile des racines appelée *rstack*. Chaque nouvel état voit sa position insérée dans *rstack* et lorsqu'une transition fermante est détectée, tous les états ne pouvant pas être des racines sont dépilés. À tout moment le sommet de *rstack* contient la position de l'état qui a le plus de chance d'être la racine de la composante qui est en train d'être visitée.

La stratégie 6 présente cet algorithme dans le cadre du DFS générique. Les encadrés (verts) peuvent être ignorés pour le moment et les lignes qui diffèrent par rapport à l'algorithme de Tarjan sont indiquées par une étoile (en rouge). L'unique différence entre les structures de données manipulées par les deux algorithmes vient du nom de la pile *pstack* qui passe de *llstack* (pile des *lowlinks*) à *rstack* (pile des *racines*).

Pour chaque nouvel état, la méthode $\texttt{PUSH}_{Dijkstra}$ insère la position DFS associée dans *rstack* au moyen de la méthode `pushtransient`. Comme précédemment, un LIVE *number* est associé à cet état. Cet identifiant est uniquement utilisé dans la méthode $\texttt{UPDATE}_{Dijkstra}$ pour déterminer le nombre de racines potentielles qui doivent être dépilées. La pile *rstack* est alors réduite progressivement jusqu'à trouver la nouvelle racine. À la fin cette réduction, la méthode `pushnontransient` permet d'empiler au sommet de *rstack* la nouvelle racine. L'état de la composante fortement connexe ayant la plus basse position dans *dfs* est conservé au sommet de *rstack*. Une racine définitive ne peut être détectée que lors d'un $\texttt{POP}_{Dijkstra}$ en regardant si la position au sommet de *rstack* correspond à l'état qui est en train d'être dépilé. Comme cet stratégie intègre elle aussi l'optimisation de Nuutila et Soisalon-Soininen [61] [3] le transfert des états vers *visitedset* et vers *live* est identique à celui effectué pour l'algorithme de Tarjan.

L'algorithme proposé par Dijkstra utilise donc les mêmes structure de données que l'algorithme de Tarjan mais se distingue par sa pile *rstack* qui tend à être plus petite que la pile *llstack* puisqu'elle ne stocke que des racines. Du point de vue de la détection des composantes fortement connexes, l'algorithme de Dijkstra détecte l'ensemble des états d'une composante au moment

3. Bien que cette optimisation ait été proposée dans le cadre de l'algorithme de Tarjan elle s'intègre facilement dans l'algorithme de Dijkstra. Couvreur et al. [20] ont aussi proposé une optimisation similaire.

où la dernière transition fermante est visitée tandis que l'algorithme de Tarjan doit attendre que le dernier état soit dépilé. Cette détection nécessite un travail de fusion dans la méthode $\texttt{UPDATE}_{Dijkstra}$ pour chaque arc fermant tandis que cette opération est retardée au dépilement des états de *dfs* dans l'algorithme de Tarjan.

4.2.2 Déroulement de l'algorithme

Exemple.
La figure 4.2 reprend l'exemple de la section précédente mais l'applique à l'algorithme de Dijkstra. Le code couleur reste identique : états LIVE en vert, DEAD en gris barré, UNKNOWN en blanc, et LIVE *number* en rouge. L'évolution de la pile *rstack* est montrée pour chacune des étapes et un parallèle peut être fait avec l'évolution de la pile *llstack* de la figure 4.1.

Les étapes 1 à 3 montrent la découverte des états A, B, et C et leur insertion dans la pile *rstack*. À l'étape 4, l'état C est transféré vers *visitedset*, dépilé de *dfs* et sa position est supprimée de la pile *rstack*. Lors de la détection de la transition fermante (E, B) à l'étape 7, les éléments de la pile *rstack* sont dépilés successivement jusqu'à ce que le sommet référence la position DFS de l'état B. Il s'agit là d'une différence majeure avec l'algorithme de Tarjan qui ne tire pas parti du fait que B, D et E appartiennent à la même composante. À l'étape 8 la découverte de l'état F force l'insertion d'un nouvel identifiant dans la pile *rstack*. Cet état est alors considéré comme une racine potentielle jusqu'à la découverte de la transition fermante entre (F, D) à l'étape 9. Une fois cette transition explorée, la pile *rstack* est dépilée jusqu'à ce que l'état B soit considéré comme la racine de la composante fortement connexe formée des états B, D, E et F.

Les piles *rstack* (algorithme de Dijkstra) et *llstack* (algorithme de Tarjan) servent toutes les deux à capturer l'appartenance d'un état à une composante fortement connexe. Les choix de mise à jour sont clairement différents dans les deux algorithmes : celui de Dijkstra remonte systématiquement l'information à la racine potentielle en réduisant la taille de la pile *rstack* tandis que celui de Tarjan ne remonte l'information que lorsque tous les successeurs d'un état ont été visités. Ces différentes stratégies ont un impact sur la taille des deux piles *pstack* même si dans le pire cas leur taille est identique. Les deux piles évoluent donc parallèlement à la pile *dfs* mais la pile *rstack* tend à être plus petite puisque chaque transition fermante explorée peut réduire sa taille.

4.2.3 Le test de vacuité

L'algorithme de Dijkstra a servi de base à de nombreux tests de vacuité [1, 19, 20, 35] qui stockent les marques d'acceptation présentes des composantes fortement connexes avec les racines potentielles. La stratégie 6 en considérant les encadrés verts présente ce test de vacuité dans le cadre du DFS générique. La pile *rstack* stocke, en plus des racines potentielles, un ensemble représentant les marques d'acceptation présentes dans la composante fortement connexe. Ces marques sont accumulées lors d'un $\texttt{UPDATE}_{Dijkstra}$ et les racines potentielles intermédiaires retirées de *rstack* jusqu'à atteindre la nouvelle racine potentielle. Cette « remontée » permet alors de fusionner les ensembles d'acceptation associés aux racines potentielles sur le chemin.

```
Structures supplémentaires :
  struct Step {src : Q,     succ : 2^Δ,
               acc : 2^F,   pos : int} // Refinement of Step of Algo. 1

Variables Locales supplémentaires :
  live : stack of ⟨ Q ⟩
  livenum : map of Q ↦ ⟨p : int⟩
  visitedset : stack of ⟨ Q ⟩
  rstack : pstack of ⟨ p : int,  acc : 2^F ⟩

PUSH_Dijkstra(acc ∈ 2^F, q ∈ Q) → int
    p ← livenum.size()
    livenum.insert(⟨ q, p ⟩)
    rstack.pushtransient(dfs.size())
    dfs.push(⟨ q, succ(q),  acc,  p ⟩)

GET_STATUS_Dijkstra(q ∈ Q) → Status
    if livenum.contains(q) then
        return LIVE
    if visitedset.contains(q) then
        return DEAD
    return UNKNOWN

UPDATE_Dijkstra(acc ∈ 2^F, dst ∈ Q)
    dpos ← livenum.get(d)
    ⟨r,  a ⟩ ← rstack.pop(dfs.size() -1)
    a ← a ∪ acc
    while dpos < dfs[r].pos do
        ⟨r, la⟩ ← rstack.pop(r - 1)
        a ← a ∪ dfs[r].acc ∪ la
    rstack.pushnontransient(r, a)
    if a = F then
        report Accepting cycle detected !

POP_Dijkstra(s ∈ Step)
    dfs.pop()
    if rstack.top(s.pos) = dfs.size() then
        rstack.pop(dfs.size())
        // Mark this SCC as Dead.
        livenum.remove(s.src)
        visitedset.insert(s.src)
        while livenum.size() > s.pos do
            q ← live.pop()
            livenum.remove(q)
            visitedset.insert(q)
    else
        live.push(s.src)
```

Stratégie 6: Calcul des SCCs et test de vacuité basé sur l'algorithme de Dijkstra

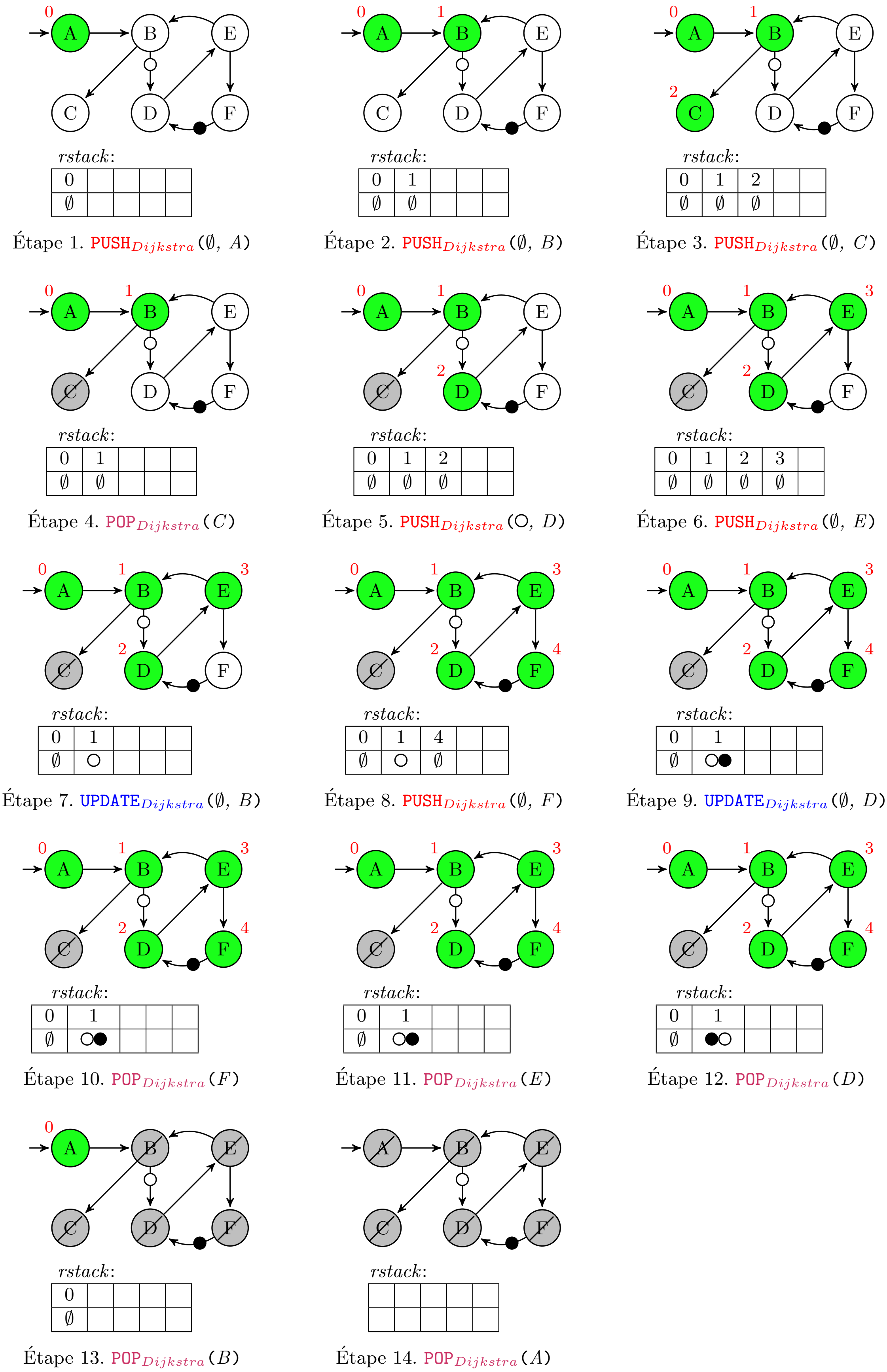


Figure 4.2 – Calcul des composantes fortement connexes et test de vacuité basé sur l'algorithme de Dijkstra

Exemple.
Cette remontée d'information permet de détecter un contre-exemple dès que toutes les transitions formant un cycle ont été vues. La figure 4.2 montre qu'un cycle acceptant peut être détecté dès l'étape 9 alors que l'algorithme de Tarjan ne peut pas détecter cela avant l'étape 12 comme le montre la figure 4.1. L'algorithme s'arrête alors et les autres étapes ne sont présentes que pour illustrer le calcul des composantes fortement connexes.

4.3 Comparaison des deux approches

En fonction des automates sur lesquels ils travaillent, les algorithmes peuvent détecter plus ou moins rapidement la présence de cycles acceptants. La figure 4.3 présente le pire cas pour les deux tests de vacuité présentés dans ce chapitre. L'automate de gauche présente le pire cas pour le test de vacuité inspiré de l'algorithme Tarjan car quelque soit l'ordre de parcours, le sous graphe représenté par le nuage sera complètement exploré avant que l'état 1 soit dépilé et qu'un contre-exemple soit détecté. L'automate de droite présente le pire cas pour le test de vacuité inspiré de l'algorithme de Dijkstra puisque si la transition fermante $(m, 0)$ est explorée avant la transition (m, n) l'ensemble des états représentés par les pointillés doivent être « fusionnés » avant d'explorer l'état n menant au contre-exemple. Nous montrerons au chapitre 6 que ces algorithmes ont néanmoins des performances très similaires.

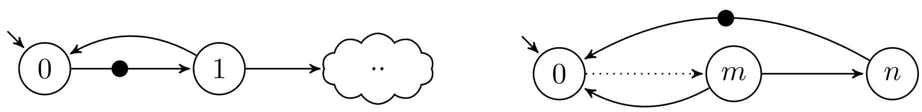
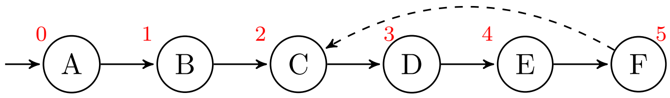


Figure 4.3 – Pires cas pour la détection des cycles acceptants : à gauche pour Tarjan, à droite pour Dijkstra.

4.4 Pile des positions compressée

Les deux algorithmes précédents intègrent l'optimisation de Nuutila et Soisalon-Soininen [61] qui réduit la taille de la pile *live* en n'y insérant pas les racines des composantes fortement connexes. La conséquence directe est que les états *transients* n'y sont donc jamais intégrés. Un état *transient* est un état qui appartient à une composante fortement connexe composée d'un unique état et d'aucune transition. Dans la pratique, les automates du manipulés sont majoritairement composés d'états transients (cf. section 6.1.3) : cette optimisation a un réel impact sur la consommation mémoire.

Un idée similaire (compatible avec l'optimisation de Nuutila et Soisalon-Soininen) peut être appliquée en observant l'évolution des piles *pstack* des algorithmes précédents. Lors de l'exploration de l'automate, la méthode `pushtransient` utilise la pile *rstack* pour insérer des positions DFS dont les valeurs sont successives et qui sont associées à un ensemble d'acceptations vide. Ces états sont alors considérés comme transients jusqu'à la détection d'une transition fermante. La pile *llstack* utilise cette méthode pour insérer des LIVE *numbers* qui sont eux aussi successifs et associé à un ensemble d'acceptation vide. Chaque appel à la méthode `pushnontransient` peut « casser » cette suite croissante de positions. Les étapes 1 à 3 des figures 4.1 et 4.2 montrent



Avant exploration (F, C)

rstack :

0	1	2	3	4	5
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

rstack (compressée) :

0					
$\emptyset$					
$\top$					

llstack :

0	1	2	3	4	5
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

llstack (compressée) :

0					
$\emptyset$					
$\top$					

Après exploration (F, C)

rstack :

0	1	2			
$\emptyset$	$\emptyset$	$\emptyset$			

rstack (compressée) :

0	2				
$\emptyset$	$\emptyset$				
$\top$	$\bot$				

llstack :

0	1	2	3	4	2
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

llstack (compressée) :

0	2				
$\emptyset$	$\emptyset$				
$\top$	$\bot$				

Figure 4.4 – Évolution de la pile compressée

```
Structures requises :
   stack_ : stack of ⟨ pos : int, acc : 2^F, is_transient : bool ⟩

pushtransient(p ∈ int)
    if stack_.is_empty() ∨ ¬ stack_.top().is_transient then
        stack_.push(⟨ p, ∅, ⊤ ⟩)

pushnontransient(p ∈ int, acc ∈ 2^F)
    stack_.push(⟨ p, acc, ⊥ ⟩)

pop(p ∈ int) → ⟨ int, 2^F ⟩
    if ¬ stack_.top().is_transient then
        ⟨ a, b, _ ⟩ ← stack_.pop()
        return ⟨ a, b ⟩
    else
        if p = stack_.top().pos then
            stack_.pop()
        return ⟨ p, ∅ ⟩

top(p ∈ int) → ⟨ int, 2^F ⟩
    if ¬ stack_.top().is_transient then
        ⟨ a, b, _ ⟩ ← stack_.top()
        return ⟨ a, b ⟩
    else
        return ⟨ p, ∅ ⟩
```

Algorithme 2: Compression de la pile des positions – Définition de la pile *pstack*.

cette insertion d'éléments consécutifs tandis que l'étape 4 montre comment l'appel à un `pushnontransient` interrompt ce schéma.

De manière plus générale, tant que des états transients sont insérés sur la pile *pstack* les positions associées sont consécutives et strictement croissantes, et la marque d'acceptation est vide. Ce constat peut être exploité pour réduire la taille de cette pile. Ainsi les éléments consécutifs peuvent être stockés au travers d'une unique entrée dans la pile *pstack* si l'on est capable de distinguer les positions référençant les états transients de celles référençant des états non-transients. Pour cela seul le premier état d'une séquence d'éléments consécutifs est stocké et un booléen *is_transient* est associé à chaque entrée de la pile. Ainsi une entrée ayant cette variable à $\top$ peut représenter plusieurs éléments. Pour récupérer l'élément au sommet (ou le dépiler) un argument supplémentaire est requis. Cet argument a un sens différent dans les deux algorithmes : pour Tarjan il s'agit du LIVE *number* de l'élément au sommet de la pile *dfs* tandis que pour Dijkstra il s'agit de la position DFS de l'élément au sommet de la pile *dfs*. Dans les deux cas l'information est stockée par la variable *dfs* et peut donc être passée en argument des méthodes `top` et `pop` sans surcoût. Dans le cas où le champ *is_transient* de l'élément au sommet est à $\perp$, cet élément est retourné directement.

L'algorithme 2 présente cette compression. Pour implémenter cette pile il est nécessaire de stocker des triplets : *pos* permettant de stocker une position, *acc* permettant de stocker une marque d'acceptation, et enfin un booléen *is_transient* indiquant si l'entrée est associée à un état transient ou non et qui permet de compresser ces éléments. La méthode `pushtransient` ajoute un élément au sommet de la pile si l'élément du sommet n'est pas transient. La méthode `pushnontransient` ajoute systématiquement une nouvelle entrée au sommet de la pile avec le champ *is_transient* positionné à $\perp$. Les méthodes `top` et `pop` ont un fonctionnement similaire et se servent de la position fournie en argument pour calculer la valeur au sommet de la pile si cet élément a le champ *is_transient* positionné à $\top$.

Exemple.
La figure 4.4 permet de visualiser l'impact de cette compression sur les deux piles avant et après détection d'un arc fermant. Lorsque la compression n'est pas utilisée on remarque que la taille des deux piles est de 6 avant la détection de l'arc fermant. Cette taille est réduite à un unique élément (pour les deux algorithmes) dans le cadre de la compression proposée ici. Après la visite de la transition fermante, les deux piles compressées sont identiques : elles ont la même taille et stockent exactement les mêmes éléments. Cette similitude masque cependant le nombre d'éléments représentés comme le montre la comparaison avec les piles originales. La compression proposée ici tend donc à la fois à réduire la consommation mémoire de l'algorithme de Dijkstra mais aussi à rendre l'algorithme de Tarjan plus compétitif en terme de mémoire.

Notons que pour les états transients le stockage de *acc* est inutile puisque toujours égal à $\emptyset$.

4.5 Conclusion

Dans ce chapitre nous avons proposé le premier test de vacuité généralisé basé sur l'algorithme de Tarjan. Ce test diffère de ceux basés sur l'algorithme de calcul des composantes fortement connexes de Dijkstra par sa gestion de la pile *pstack*. Cette différence a un impact à deux niveaux :

1. pour le test basé sur l'algorithme de Dijkstra, la détection d'un contre-exemple est faite dès que l'ensemble des transitions le constituant sont visitées ;
2. le test basé sur l'algorithme de Tarjan tend à être plus gourmand en mémoire même s'il a la même complexité au pire cas que le test basé sur l'algorithme de Dijkstra. Nous montrerons cette différence mémoire au chapitre 6.

Pour réduire le coût mémoire de ces algorithmes, nous avons aussi mis en place une pile des positions qui s'intègre naturellement dans tous les algorithmes de cette section. Cette pile vise à compresser tous les états considérés comme transients à un instant donné. Cette idée complète celle exprimée par Nuutila et Soisalon-Soininen [61] pour économiser le stockage des racines dans la pile des états vivants. Comme nous avons distingué algorithmes de calcul des composantes fortement connexes et tests de vacuité cette optimisation de la pile des positions est entièrement applicable dans les deux cas.

L'intégration de toutes ces optimisations permet de distinguer deux types de tests de vacuité généralisés. Les premiers basés sur l'algorithme de Dijkstra semblent mieux adaptés en phase de debug du modèle puisqu'ils permettent une détection au plus tôt des contre-exemples. Les seconds semblent adaptés à la vérification de modèles sûr puisque la détection des cycles acceptants est retardée. Les performances relatives de ces deux familles d'algorithme seront comparées au chapitre 6.

Chapitre 5

Tests de vacuité basés sur un union-find

How can one check a routine in the sense of making sure that it is right ?

Alan M. Turing

Sommaire

Les tests de vacuité basés sur le calcul des composantes fortement connexes ne génèrent qu'une unique fois les successeur d'un état à la différence des NDFS *qui peuvent les générer deux fois dans le pire cas. Cet argument masque la réalité : lorsqu'une composante est détectée, tous ses états doivent être reparcourus pour être marqués morts. Ce second parcours est cependant moins coûteux puisqu'il ne sollicite pas la fonction de transition. Pour remédier à cela, Gabow [30] suggère l'utilisation d'une structure appelé union-find. Cette proposition n'a jamais été exploitée ni pour le calcul des composantes fortement connexes ni pour les tests de vacuité.*

Ce chapitre propose donc de palier cela en commençant par décrire cette structure (section 5.1) puis en montrant comment elle peut être combinée avec les tests de vacuité généralisés. Enfin la section 5.2.3 s'intéresse à la compatibilité d'une telle approche avec d'autres optimisations telles que le Bit State Hashing *et le* State Space Caching.

5.1 L'union-find

Éloignons-nous du problème de la vérification de systèmes au moyen d'automates le temps d'expliquer le fonctionnement de la structure d'union-find [83].

5.1.1 Description de la structure

Pour résoudre un problème de compilation d'instructions Fortran, Galler et Fisher [32] proposent en 1964 l'utilisation d'une structure dédiée aux calculs des classes d'équivalences d'un ensemble non-vide S. La formulation du problème peut être résumée de la manière la suivante :

« Comment une structure peut elle efficacement permettre la création de nouvelles classes d'équivalences, leur fusion et tester l'appartenance de deux éléments à la même classe ? »

Pour répondre à cette question Galler et Fisher ont proposé une interface simple appelée *union-find* en raison des opérations permettant de la manipuler :

- `makeset` (x) : permet la création d'une nouvelle classe contenant uniquement l'élément passé en argument ;
- `unite` (x, y) : permet d'unir les deux classes associées aux deux éléments passés en paramètres ;
- `find` (x) : permet de récupérer le représentant de la classe d'équivalence d'un élément passé en paramètre.

Plus précisément si S est un ensemble non-vide de n éléments distincts, et S_1 et S_2 deux sous-ensembles de S, on dit que S_1 et S_2 sont disjoints si $S_1 \cap S_2 = \emptyset$. La structure d'union-find maintient une collection dynamique d'ensembles disjoints $\{S_1, \ldots, S_k\}$, avec S_i un sous-ensemble de S et $1 \leq i \leq k \leq n$. Chaque classe est associée à un représentant unique qui est généralement un des éléments de cette classe et qui peut être modifié à chaque opération `find` ou `unite`. Cette structure est généralement représentée sous la forme d'un tableau dans lequel chaque élément est une entrée qui pointe vers son représentant.

L'algorithme 3 montre un cadre d'utilisation classique cette structure. Les classes sont tout d'abord construites à l'aide de l'opération `makeset` (lignes 4 et 5) puis, l'ensemble des unions à réaliser (ici exprimée par *unionset*) est parcourue pour effectuer toutes les unions requises. Si deux éléments ne sont pas dans la même classe alors leurs classes doivent être unies (ou jointes), sinon l'union peut être ignorée (ligne 7 et 8).

```
1 Input:   S : (non-empty) set of distincts elements
2          uf : union-find of ⟨ S ⟩
3          unionset : set of ⟨ left ∈ S, right ∈ S ⟩

4 foreach s ∈ S do
5     uf.makeset(s)

6 foreach lr ∈ unionset do
7     if uf.find(lr.left) ≠ uf.find(lr.right) then
8         uf.unite(lr.left, lr.right)
```

Algorithme 3: Exemple d'utilisation de l'union-find.

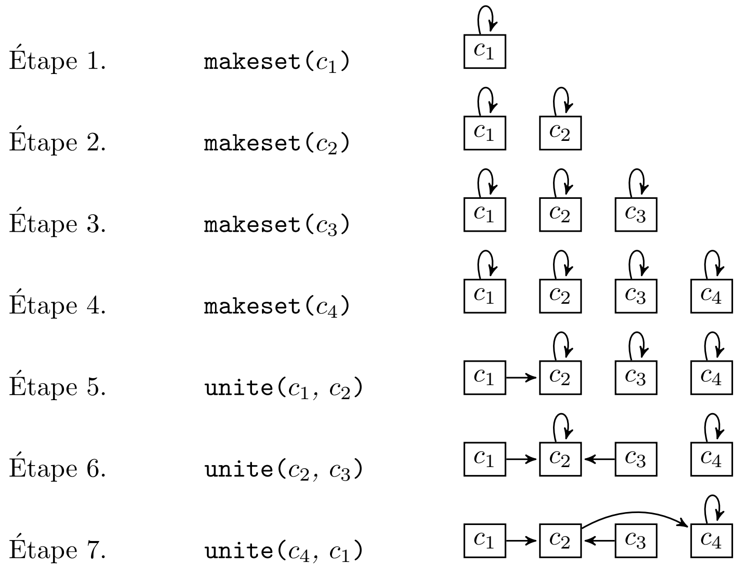


FIGURE 5.1 – Déroulement d'une exécution de l'algorithme 3 avec $S = \{c_1, c_2, c_3, c_4\}$ sur une structure d'union-find.

Exemple.
La figure 5.1 permet de visualiser une évolution possible de cette structure sur un exemple simple pour $S = \{c_1, c_2, c_3, c_4\}$. Les quatre premières étapes montrent la création des partitions associées à chaque élément de S (en accord avec les lignes 4–5 de l'algorithme 3). À chaque nœud est attribué un *lien de parenté*. Lorsque la source et la cible de ce lien sont identiques, le nœud est son propre représentant. Sinon, il faut « remonter » les liens de parenté jusqu'à trouver le représentant de la classe au moyen de la méthode `find`. Lors d'une opération `makeset` l'élément inséré est nécessairement son propre représentant. Les liens de parenté peuvent être modifiés lors d'une opération `unite` qui permet l'union de deux classes. Pour cela il suffit de faire pointer le parent d'une des deux classes vers le parent de l'autre comme c'est le cas dans les étapes 5 à 7. Ainsi chaque opération `unite` utilise l'opération `find`.

5.1.2 Optimisations

Une structure d'union-find est essentiellement sollicitée pour rechercher les représentants des classes qu'elle contient. Un des critères d'efficacité est donc la minimisation de la distance entre un nœud et son représentant.

Exemple.
Par exemple, l'étape 7 de l'exemple figure 5.1 serait optimal (pour des opérations `find` futures) si les éléments c_1, c_2, et c_3 étaient à une distance de 1 de l'élément c_4. Maintenir une telle distance est coûteux puisqu'il faut parcourir tous les éléments de la classe pour mettre à jour les liens de parenté.

Plusieurs optimisations ont été proposées pour minimiser cette distance :

`QUICK` Cette optimisation tire parti de la redondance des calculs entre les opérations `find` et `unite` : stocker les résultats de l'opération `find` (ligne 7 de l'algorithme 3) permet de passer directement les représentants de chaque classe à l'opération `unite`. Ainsi cette opération évite de recalculer les représentants de chaque classe.

`PATH COMPRESSION` Cette optimisation vise à compresser les chemins lors de chaque opération `find`. À la fin de cette opération tous les nœuds sur le chemin de parenté sont à une distance de 1 du représentant de la classe à laquelle ils appartiennent. Dans cette optimisation, certains nœuds peuvent être très éloignés de la racine mais une seule opération `find` permet de le remonter à une distance de 1.

`IMMEDIATE PARENT CHECK` Cette optimisation vient compléter les deux optimisations précédentes en testant si les deux éléments ont le même parent avant d'effectuer une union (en utilisant la technique `QUICK`). Avec l'optimisation `PATH COMPRESSION` une grande partie des chemins est compressée et la probabilité que deux nœud dans la même partition aient le même parent est forte.

`LINK by RANK` Cette optimisation diffère des optimisations précédentes en proposant une stratégie pour équilibrer l'arbre enraciné formé par tous les éléments d'une classe. L'idée est d'associer à chaque nœud un poids initialement à 0. Lors d'une opération `unite`, si les poids des deux représentants sont égaux, le nouveau représentant est choisi aléatoirement parmi ces deux représentants et son poids est incrémenté. Sinon c'est le représentant ayant le plus petit poids qui modifie son parent pour pointer vers l'autre représentant.

`MEMORY SMART` Cette optimisation propose de réduire la consommation mémoire des algorithmes. Si l'on combine toutes les optimisations précédentes, chaque nœud stocke : un élément, un poids, et un lien de parenté. Comme le poids n'est utile que pour les représentants et que ces derniers sont leurs propres parent l'idée est de fusionner ces deux champs. Lorsque les éléments stockés sont des entiers, il est facile de représenter l'union-find sous forme d'un tableau : un parent peut être référencé par son indice tandis que les poids peuvent être stockés négativement. Ainsi, un entier par nœud peut être économisé.

Les cinq optimisations présentées ci-dessus peuvent être combinées entre-elles et constituent un bon compromis entre complexité théorique et performances pratiques [62]. Les opérations `find` et `unite` ont alors une complexité inversement proportionnelle à la fonction d'Ackermann ($Ack(n)$) qui est connue pour croître très lentement (plus le nombre d'éléments n augmente).

5.2 Tests de vacuité avec union-find

L'union-find est une structure adaptée au partitionnement d'un ensemble. Dans le cadre des tests de vacuité, l'idée de partitionner les états en fonction de la composante fortement connexe à laquelle ils appartiennent vient naturellement. Comme ces tests doivent distinguer les états vivants des états morts, une partition spéciale contenant un état artificiel nommé *alldead* peut être créée : marquer un ensemble d'états comme morts peut être fait en une opération et revient à unir la partition contenant ces états avec la partition contenant *alldead*. Le statut d'un état peut alors être détecté simplement : il est inconnu s'il n'est pas présent dans l'union-find, il est mort s'il est dans la même partition que *alldead*, il est vivant sinon. Deux nouvelles méthodes peuvent donc être introduites :

`sameset` (x, y) : permet de tester si les deux états passés en paramètres (x et y) sont dans la même partition. Cela peut être fait efficacement en comparant les représentants des deux classes associées et il s'agit donc simplement d'un raccourci d'écriture pour `find` (x) = `find` (y) ;

`contains` (x) : permet de tester la présence d'un élément dans la structure.

L'objectif de ce chapitre est d'adapter les tests de vacuité du chapitre précédent pour utiliser une structure d'union-find qui va remplacer les variables *live*, *livenum* et *visitedset* et ainsi permettre le marquage d'une composante comme morte en une unique opération.

5.2.1 Test de vacuité avec union-find basé sur l'algorithme de Tarjan

L'algorithme présenté ici s'inspire de celui de Tarjan : les *lowlinks* sont stockés dans la pile *llstack* et sont mis à jour, soit lorsqu'un état est dépilé de *dfs*, soit lors de la détection d'une transition fermante. L'union-find stocke seulement l'appartenance d'un état à une composante fortement connexe, et deux états sont unis s'ils appartiennent à la même composante (opération `unite`). Comme cette union ne peut être faite que si les deux états sont déjà présents dans l'union-find, tout nouvel état y est automatiquement inséré : le nombre d'appels à `makeset` est donc exactement le nombre d'états explorés de l'automate. Enfin, dès qu'une composante fortement connexe est détectée, si celle-ci ne contient pas de cycles acceptants, il faut la marquer comme morte. Ce marquage se fait simplement par l'union de sa racine avec *alldead*. Savoir si un état q est mort revient alors à tester `sameset(`q, *alldead*`)`.

Dans l'algorithme de Tarjan (détails section 4.1.3) chaque état est associé à un LIVE *number* qui est attribué en fonction du nombre d'états vivants en cours. Ce nombre peut facilement être maintenu par la structure d'union-find et la méthode `makeset` peut être modifiée pour retourner le LIVE *number* de l'état qui est inséré. Nous définissons une méthode `liveget` pour connaître le LIVE *number* d'un état. La section 5.2.3 décrit comment ces opérations peuvent être implémentées pour être en temps constant.

La stratégie 7 présente ce nouveau test de vacuité. Les encadrés (roses) permettent de distinguer calcul des composantes fortement connexes et test de vacuité. L'union-find est initialisé avec la partition *alldead* ligne 8. Chaque appel à PUSH$_{Tarjan-uf}$ utilise la méthode `makeset` pour insérer le nouvel état et récupère son LIVE *number* qui est ensuite empilé sur *llstack*. La méthode UPDATE$_{Tarjan-uf}$ n'est modifiée par rapport à la stratégie originale que par l'ajout de la ligne 32 : l'état source et l'état destination de la transition fermante sont alors regroupés dans la même partition de l'union-find. De la même manière lorsqu'un état qui n'est pas une racine est dépilé de la pile *dfs* dans la méthode POP$_{Tarjan-uf}$ une union est faite entre cet état et son prédécesseur dans la pile *dfs*. Au fur et à mesure que les états sont dépilés ils sont ajoutés à la partition représentant la composante fortement connexe qui est en train d'être construite. Lorsque la racine est détectée, un simple appel à `unite` permet de marquer toute la composante comme morte (ligne 30). Il est clair au regard de cette opération que l'énumération des états de la composante fortement connexe n'est pas immédiate puisque cette information est masquée par l'union-find. Cette restriction n'est pas gênante dans le cadre du *model checking*.

```
Structures supplémentaires :
  struct Step {src : Q,     succ : 2^Δ,
               acc : 2^F,   pos : int} // Refinement of Step Algo. 1
Variables Locales supplémentaires :
  uf : union-find of ⟨ Q ∪ {alldead} ⟩
  llstack : pstack of ⟨ p : int, acc : 2^F ⟩
Initialisation :
  uf.makeset(alldead)

PUSH_{Tarjan-uf}(acc ∈ 2^F, q ∈ Q) → int
  p ← uf.makeset(q)
  llstack.pushtransient(p)
  dfs.push(⟨ q, succ(q), acc, p ⟩)

GET_STATUS_{Tarjan-uf}(q ∈ Q) → Status
  if uf.contains(q) then
    if uf.sameset(q, alldead) then
      return DEAD
    return LIVE
  return UNKNOWN

UPDATE_{Tarjan-uf}(acc ∈ 2^F, dst ∈ Q)
  ⟨ p, a ⟩ ← llstack.pop(dfs.top().pos)
  llstack.pushnontransient( min(p, uf.liveget(d)), acc ∪ a )
  uf.unite(dst, dfs.top().src)
  if acc ∪ a = F then
    report Accepting cycle detected !

POP_{Tarjan-uf}(s ∈ Step)
  dfs.pop()
  ⟨ ll, acc ⟩ ← llstack.pop(s.pos)
  if ll = s.pos then
    // Mark this SCC as dead.
    uf.unite(s.src, alldead)
  else
    uf.unite(s.src, dfs.top().src)
    ⟨ p, a ⟩ ← llstack.pop(dfs.top().pos)
    llstack.pushnontransient( min(p, ll), a ∪ acc ∪ s.acc )
    if a ∪ acc ∪ s.acc = F then
      report Accepting cycle detected !
```

Stratégie 7: Test de vacuité basé sur l'algorithme de Tarjan avec union-find.

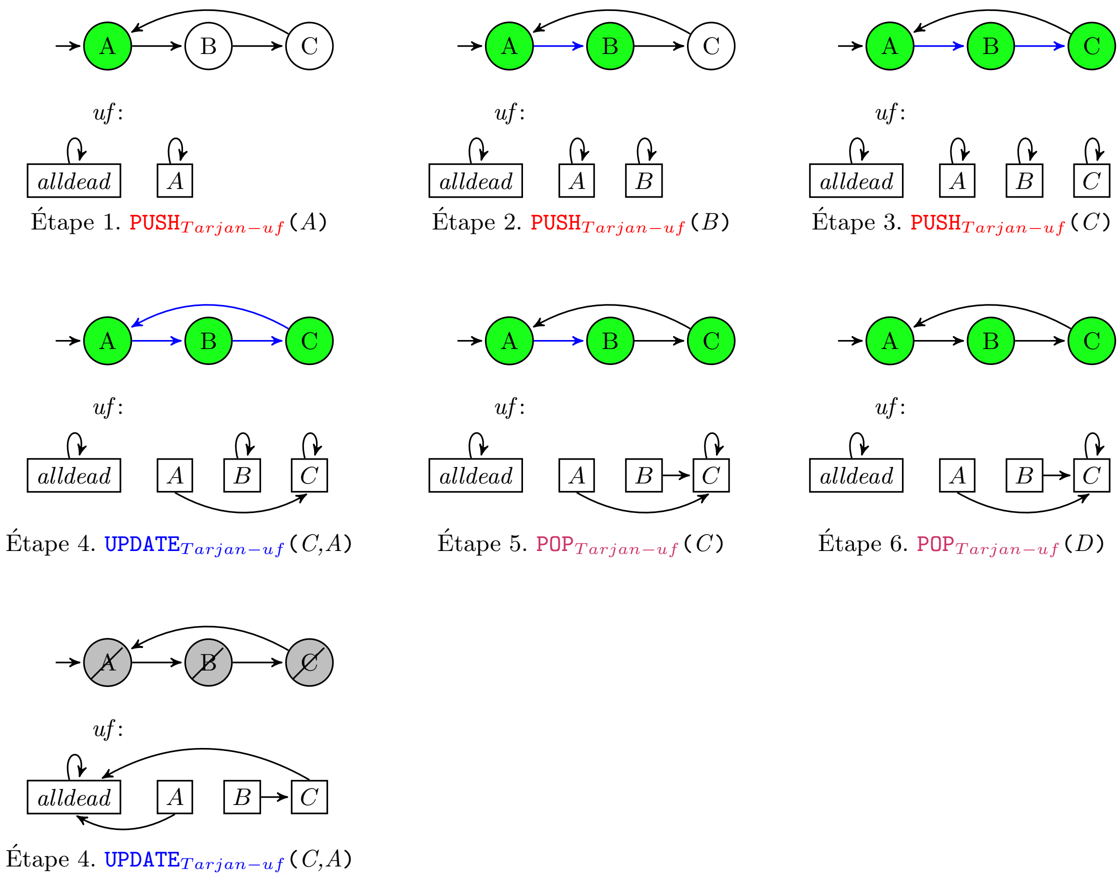


FIGURE 5.2 – Déroulement de l'adaptation de Tarjan utilisant un union-find.

Exemple.
La figure 5.2 présente l'évolution de la structure d'union-find lors de cette stratégie sur un exemple simple. Dans les étapes 1 à 3 les états A, B, et C sont découverts, et les partitions associées sont créées au sein de l'union-find. Lors de la découverte de la transition fermante à l'étape 4, les partitions {A} et {C} sont réunies et {C} est choisi comme représentant. À l'étape 5 dépiler l'état C de *dfs* conduit à unir {B} à l'ensemble {C, A}. Lorsque l'état B est dépilé il fait déjà partie de la même partition que l'état A, l'opération d'union est sans effet sur la structure d'union-find. Enfin, la dernière étape montre la détection de la racine de la SCC, on effectue donc une opération `unite(`A, *alldead*`)` : cette composante est alors considérée comme morte. Notons que dans cet exemple, il y a une compression de chemin au sein de l'union-find et l'état A est à une distance de un de la racine. L'état B n'est pas sur le chemin de parenté son lien de parenté n'est donc pas modifié et il est à une distance de deux du représentant.

5.2.2 Test de vacuité avec union-find basé sur l'algorithme de Dijkstra

L'algorithme présenté à la section précédente est dans « l'esprit de l'algorithme de Tarjan » : il utilise la notion de LIVE *numbers* pour la détection des racines des composantes fortement connexes. Cette inspiration a nécessité la mise en place d'une structure d'union-find étendue pour récupérer le LIVE *number* d'un état. La construction d'un test de vacuité à base d'un

```
Structures supplémentaires :
  struct Step {src : Q,    succ : 2^Δ,
               acc : 2^F }        // Refinement of Step of Algo. 1
Variables Locales supplémentaires :
  uf : union-find of ⟨ Q ∪ {alldead} ⟩
  rstack : pstack of ⟨ p : int, acc : 2^F ⟩
Initialisation :
  uf.makeset(alldead)

PUSH_Dijkstra-uf(acc ∈ 2^F, q ∈ Q) → int
  uf.makeset(q)
  rstack.pushtransient(dfs.size())
  dfs.push(⟨ q, succ(q), acc ⟩)

GET_STATUS_Dijkstra-uf(q ∈ Q) → Status
  if uf.contains(q) then
    if uf.sameset(q, alldead) then
      return DEAD
    return LIVE
  return UNKNOWN

UPDATE_Dijkstra-uf(acc ∈ 2^F, dst ∈ Q)
  ⟨r, a⟩ ← rstack.pop(dfs.size() -1)
  a ← a ∪ acc
  while ¬ uf.sameset(d, dfs[r].src) do
    uf.unite(d, dfs[r].src)
    ⟨r, la⟩ ← rstack.pop(r - 1)
    a ← a ∪ dfs[r].acc ∪ la
  rstack.pushnontransient(r, a)
  if a = F then
    report Accepting cycle detected !

POP_Dijkstra-uf(s ∈ Step)
  dfs.pop()
  if rstack.top(s.pos) = dfs.size() then
    rstack.pop(dfs.size())
    // Mark this SCC as Dead.
    uf.unite(s.src, alldead)
```

Stratégie 8: Test de vacuité basé sur l'algorithme de Dijkstra avec union-find.

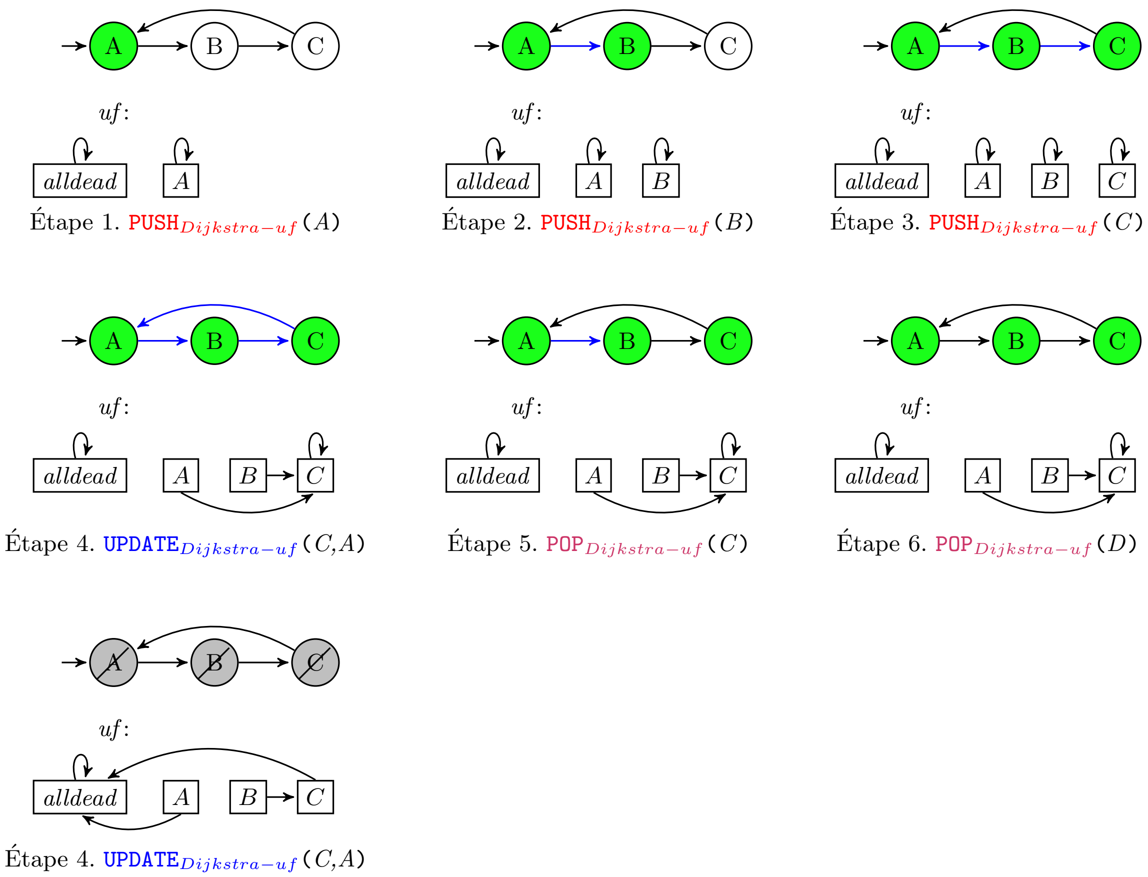


Figure 5.3 – Déroulement de l'adaptation de Dijkstra utilisant un union-find.

union-find « classique » est néanmoins possible [72] en utilisant une pile des racines comme proposé dans l'algorithme de Dijkstra. Comme cette pile des racines est basée sur des positions DFS il n'est pas nécessaire d'équiper l'union-find pour qu'il gère les LIVE *number*.

Ce nouveau test de vacuité est présenté stratégie 8. Pour les mêmes raisons que la stratégie de la section précédente, les états sont insérés dans l'union-find à chaque appel à PUSH$_{Dijkstra-uf}$. Le reste de l'algorithme colle de très près à l'algorithme de Dijkstra : les unions sont effectuées lors de la fusion de la pile des racines dans une opération UPDATE$_{Dijkstra-uf}$ et tous les états sont marqués comme morts en une opération lors d'un POP$_{Dijkstra-uf}$. Dans la méthode UPDATE$_{Dijkstra-uf}$, tester si deux états sont déjà dans la même partition suffit car l'algorithme de Dijkstra fusionne toutes les partitions au fur et à mesure que les transitions fermantes sont détectées. La figure 5.3 applique cette stratégie à l'exemple de la section précédente. On voit que la partition contient l'intégralité des états de la composante fortement connexe dès l'étape 4 par opposition à l'étape 5 pour la stratégie précédente.

Note : Dans les algorithmes originaux de Tarjan et de Dijkstra et pour un ordre de parcours donné, les structures de données manipulées sont strictement identiques à l'exception de la pile *pstack*. Comme la structure d'union-find est sensible à l'ordre des unions, cette propriété n'est plus nécessairement vraie pour les algorithmes à base d'union-find : pour une étape donnée, les deux structures d'union-find peuvent avoir des structures différentes à cause des unions publiées qui sont différentes.

5.2.3 Compatibilité avec les techniques de réduction

Pour que les algorithmes à base d'union-find soient compétitifs avec les autres algorithmes il est nécessaire qu'ils soient compatibles avec les principales techniques permettant de combattre la consommation mémoire. Cette section montre aussi comment le maintien des LIVE *numbers* peut être effectué pour être compatible avec l'algorithme présenté section 5.2.1.

Compatibilité « à la volée » Traditionnellement, la structure d'union-find est utilisée sur un ensemble non-vide S dont les éléments sont connus à l'avance. Ces éléments peuvent alors être numérotés de 0 à $|S| - 1$ et un simple vecteur d'entiers V de taille $|S| - 1$ initialisé à -1 permet le stockage de la relation de parenté : pour connaître le représentant associé à un élément s de S il suffit de regarder la valeur en $V[x_s]$, avec x_s l'identifiant associé à s. Si cette valeur est égale à -1, l'état s est son propre représentant, sinon il s'agit d'une référence vers une autre case du vecteur et l'opération peut être recommencée jusqu'à trouver -1.

Dans le cas qui nous intéresse, les éléments sont insérés dynamiquement par `makeset` et l'ensemble des états n'est pas connu à l'avance. Pour palier cela l'union-find peut être implémenté au moyen d'un vecteur et d'une table de hachage. À chaque insertion d'un nouvel état s un compteur global (initialement à 0) est incrémenté et le couple (état, compteur) est inséré dans la table de hachage. Le vecteur est alors augmenté d'une case qui est affectée à -1. Ainsi, l'identification des éléments est faite de manière dynamique. Dans ce cas, les opérations de haut niveau (`makeset`, `unite`) manipulent des états mais l'union-find utilise des entiers comme représentants des partitions.

Compatibilité *Bit State Haching* et *State Space Caching* Les algorithmes basés sur le calcul des composantes fortement connexes ne peuvent appliquer ces deux techniques que sur les états morts (même partition que *alldead*). Lors d'une opération `unite(`s, *alldead*`)`, l'état s est nécessairement une racine et son identifiant est plus petit que les identifiants associés aux états vivants qu'il peut atteindre. Dans le vecteur de parenté, tous les états ayant une position supérieure ou égale à x_s peuvent donc être transférés dans un ensemble *visitedset* compatible avec le *Bit State Haching* et le *State Space Caching*. L'utilisation d'une table de hachage bidirectionnelle permet ce transfert de manière efficace. Le compteur utilisé lors d'une opération `makeset` peut alors être réaffecté à x_s. Ce compteur garde donc le nombre d'états vivants dans la structure d'union-find. Nous notons ce compteur *live_*.

Cependant transférer tous les états ayant une position supérieure à x_s revient à supprimer l'avantage offert par l'union-find à savoir marquer un ensemble d'états comme morts en une opération. Cet avantage peut être conservé en retardant le transfert des états morts dans *visitedset*. Pour cela aucun transfert n'est effectué lors d'une union avec *alldead*, mais le compteur *live_* est mis-à-jour comme expliqué précédemment. Lors d'une opération `makeset(`q_1`)`, s'il existe un état ayant la valeur de *live_* il peut être transféré dans *visitedset*, et supprimé de la table de hachage bidirectionnelle. Ensuite l'état q_1 peut prendre la valeur de *live_* et ce compteur peut être incrémenté.

Avec cette structure d'union-find la gestion des LIVE *number* est possible : lors d'un `makeset` il suffit de retourner la valeur associée à l'état, et la méthode `liveget` retourne seulement l'identifiant associé à un état dans la table de hachage bidirectionnelle. Pour un état donné, savoir s'il est mort revient à tester d'abord sa présence dans l'union-find. S'il est présent mais associé à un identifiant strictement supérieur à *live_* il est mort. Sinon il faut tester sa présence dans *visitedset*. Ici *alldead* devient alors juste un subterfuge logique, permettant de capturer la « mise à mort » d'une composante fortement connexe.

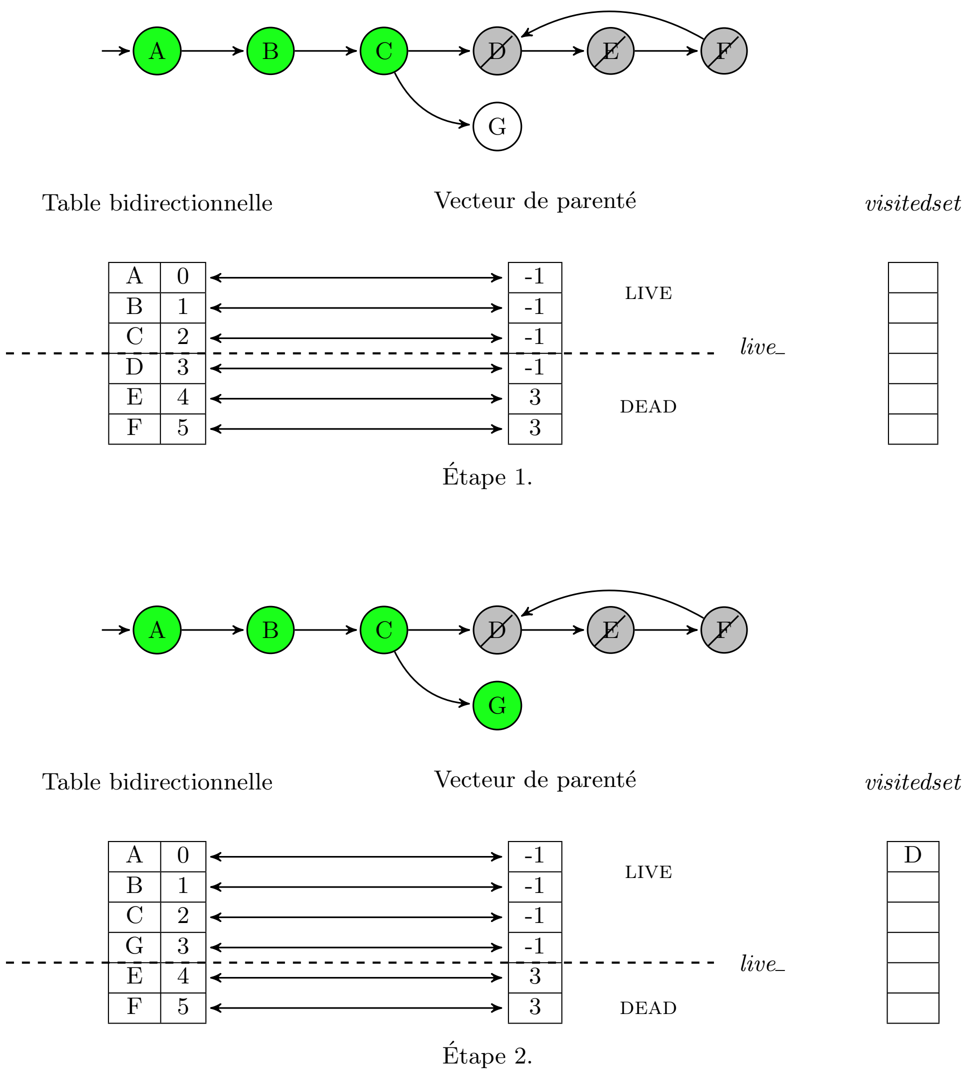


FIGURE 5.4 – Compatibilité *Bit State Hashing* et maintient du *LIVE number*

Exemple.
La figure 5.4 présente l'évolution de cette structure pendant l'exploration d'un automate. Les états verts représentent les états vivants, les états barrés (gris) ceux morts, et les états blancs ceux qui n'ont pas encore été visités. Dans cet exemple, l'étape 1 présente un automate dans lequel les états {D, E, F} viennent d'être marqués comme morts. Le compteur *live_* scinde alors le vecteur de parenté en deux avec une partie contenant les vivants et une partie contenant les morts. Ainsi comme le LIVE *number* de l'état D est supérieur à *live_* cet état est mort. L'étape 2 montre l'insertion du nouvel état G. Celui-ci prend la place de l'état D. Pour tester si l'état D est vivant il est maintenant nécessaire de tester sa présence dans la table bidirectionnelle puis de tester sa présence dans *visitedset*.

Note : ce transfert retardé permet d'éviter la suppression d'états dans la table bidirectionnelle et l'ajout d'état dans *visitedset* qui peuvent être coûteux. De plus, cette structure particulière d'union-find est compatible avec toutes les techniques d'optimisation présentées en section 5.1.2 puisqu'elles ne s'appliquent que sur le vecteur de parenté.

5.3 Conclusion

Dans ce chapitre nous avons vu que les tests de vacuité basés sur le calcul des composantes fortement connexes nécessite d'énumérer tous les états d'une composante lorsque celle-ci est détecté. Pour palier cette énumération nous avons proposé l'utilisation d'une structure d'union-find qui permet de marquer l'intégralité de la composante comme ayant été visitée en une unique opération.

Cette structure d'union-find s'intègre donc parfaitement dans les algorithmes de calcul de composantes fortement connexes de Tarjan et de Dijkstra. Nous avons montré comment cette intégration pouvait être faite et comment des tests de vacuité généralisés pouvaient être construit à partir de ces nouveaux algorithmes. Ces algorithmes peuvent alors bénéficier des nombreuses optimisations existantes sur les structures d'union-find : nous en avons présenté les principales mais d'autres, pouvant impacter les performances des tests de vacuité, existent. L'analyse de cet impact constituerait une étude intéressante.

L'utilisation d'une structure d'union-find permet aussi une simplification des algorithmes puisque les structures permettant de connaître le statut d'un état (durant le parcours DFS) sont fusionnées. Nous pouvons aussi noter que les tests de vacuité présentés dans ce chapitre s'intègrent parfaitement dans le cadre du DFS générique présenté en début de manuscrit.

Enfin, les tests de vacuité présenté ici s'attaquent à une catégorie d'automate peu étudiée : les automates généralisés. Il est notoire que ces automates sont complexes à vérifier et très peu de travaux s'intéresse à proposer des techniques permettant une vérification efficace. Cela est d'autant plus dommage qu'ils peuvent permettre une expression de l'équité à moindre coût et, le chapitre suivant montre l'impact lié à l'utilisation d'automates non-généralisés.

Chapitre 6

Comparaison des algorithmes séquentiels

Errors using inadequate data are much less than those using no data at all.

Charles Babbage

Sommaire

Dans cette partie, nous avons montré qu'il existe de nombreux tests de vacuité pour les automates forts. Nous nous sommes particulièrement intéressés aux tests basés sur le calcul des composantes fortement connexes qui gèrent naturellement les automates généralisés. Au chapitre 4, nous avons proposé une nouvelle optimisation pour réduire la consommation mémoire de ces tests, tandis qu'au chapitre 5 nous avons suggéré deux nouveaux algorithmes. Ce chapitre vise à comparer les performances relatives de tous ces tests de vacuité.

6.1 Description du jeu de tests

L'un des jeux de test le plus utilisé pour le *model checking* explicite est celui de BEEM [1] qui est composé de modèles accompagnés de formules. Malheureusement, la majorité des formules qui le composent se traduisent en automates faibles ou terminaux. Bien que ces automates puissent être traité directement par les algorithmes présentés dans les chapitres précédents, ils ne sont pas pertinents car :

1. Benchmarks for Explicit Model Checkers. http://paradise.fi.muni.cz/beem/

1. ils ont une structure particulière qui ne sollicite pas tous les aspects des tests de vacuité ;
2. ils peuvent être traités par des tests de vacuité dédiés plus efficaces et moins coûteux en mémoire.

Cette section propose un jeu de test permettant une comparaison « juste » des différents tests de vacuité dédiés aux automates forts.

6.1.1 Modèles

Les modèles présents dans BEEM ont été répertoriés par Pelánek [64, 65, 66] en fonction du type de problème qu'ils résolvent (protocole, exclusion mutuelle, . . .) et de la structure de l'espace d'états qu'ils génèrent. Quatre familles structurelles d'espace d'états sont distinguées :

(a) l'espace d'état possède de longs cycles, i.e. il existe au moins une grosse composante fortement connexe ;

(b) l'espace d'état est composé de nombreuses composantes fortement connexes de petite taille ;

(c) l'espace d'état est composé d'une grosse composante fortement connexe dont la structure est complexe ;

(d) l'espace d'état possède la forme d'un arbre ou est linéaire, les composantes fortement connexes sont de petite taille.

Afin de construire un jeu de test qui soit le plus représentatif possible, nous avons sélectionné des modèles venant de chaque catégorie. La table 6.1 les présente et spécifie pour chacun leur type, leur nombre d'états, de transitions et de composantes fortement connexes. Trois informations supplémentaires y sont aussi mentionnées :

1. le ratio entre le nombre de transitions et le nombre d'états : c'est le degré moyen de transitions sortantes par état ;
2. le nombre d'états transients, i.e. les composantes fortement connexes composées d'un unique état sans boucle. Ce nombre est important car il indique le nombre d'états du modèle qui ne pourront pas se synchroniser pour construire des composantes fortement connexes non transientes dans le produit synchronisé ;
3. la taille (en nombre d'entiers) nécessaire au stockage d'un état. Ce nombre permet d'estimer la mémoire nécessaire au stockage de l'intégralité de l'espace d'états du modèle. Pour cela il suffit du multiplier la taille d'un état par le nombre d'états.

L'étude de ce tableau montre que :

- tous les types de modèles de la classification de Pelánek sont présents, ce qui permet d'avoir un échantillon représentatif qui va pouvoir stimuler tous les aspects des tests de vacuité ;
- le nombre moyen de transitions sortantes par état fluctue entre 1 et 7. Ce ratio donne une idée sur la structure de l'espace d'état : plus le ratio est grand plus l'espace d'état est large, plus il est petit plus l'espace d'état est profond. Ainsi le modèle *elevator2.3* est bien plus large que le modèle *adding.4* ;
- le nombre d'états transients est très important pour les modèles de type (a) et (b) puisqu'en moyenne ils représentent 85% des états du modèle. On peut même noter que 99,9% des états du modèle *leader-election* sont transients. En revanche, pour les modèles de type (c)

Modèle	Type	États	Transitions	Nombre de SCCs	Ratio trans./états	Nombre d'états transient	Taille état (int)
adding.4	(d)	3 370 680	5 683 994	3 370 680	1.68	2 887 968	5
bridge.3	(d)	838 864	2 012 020	838 864	2.39	723 817	14
brp.4	(a)	12 068 447	25 137 148	1 696 242	2.08	1 642 503	18
collision.4	(a)	41 465 543	113 148 818	92 948	2.72	92 915	22
cyclic-scheduler.3	(a)	229 374	1 597 440	16 383	6.96	16 382	52
elevator.4	(c)	888 053	2 320 984	6	2.61	5	20
elevator2.3	(c)	7 667 712	55 377 920	1	7.22	0	36
exit.3	(d)	2 356 294	7 515 084	2 356 294	3.18	1 888 542	28
leader-election.3	(b)	101 360	446 025	101 360	4.40	101 359	121
production-cell.3	(c)	822 612	2 496 342	1	3.03	0	22

TABLE 6.1 – Caractéristiques des modèles utilisés.

et (d) ce nombre est très faible (moins de 5%) ce qui implique des composantes fortement connexes avec une structure très complexe. On remarque enfin que le modèle *elevator2.3* ne possède aucun état transient ;

- le nombre d'états varie entre 1×10^5 et 4×10^7 ce qui permet d'avoir aussi bien de petits espaces d'états que des gros. Ce nombre doit être combiné avec le nombre d'entiers nécessaires au stockage d'un état. Si un entier est stocké sur 32 bits, le stockage de l'intégralité du modèle *leader-election.3* nécessite : $\frac{101\,306 \times 121 \times 32}{8} = 49$ méga-octets par exemple ;
- le nombre de composantes fortement connexes oscille entre 1 et 3×10^5 ce qui permet d'avoir des composantes fortement connexes avec des structures variables. En effet, plus le nombre de composantes est petit plus il y aura de transitions internes aux composantes. Par exemple, le modèle *elevator2.3* ne possède qu'une unique composante fortement connexe, cela signifie que toutes ses transitions sont internes à cette composante.

On constate donc que cette sélection de modèles couvre de nombreuses formes d'espaces d'états et permet d'anticiper dans une certaine mesure la structure de l'espace d'état du produit synchronisé. Par exemple, il est probable que les modèles ayant un grand nombre d'états transients généreront des produits synchronisés en ayant eux aussi beaucoup. La section suivante introduit les différentes formules qui seront utilisées pour réaliser le produit synchronisé.

6.1.2 Formules

Comme dit précédemment, les formules associées aux différents modèles de BEEM ne sont pas pertinentes pour notre cas d'étude. Pour y remédier, nous avons décidé de générer des formules aléatoires qui se traduisent en un automate de Büchi fort. Afin d'obtenir des formules qui ne soient pas triviales à vérifier, les règles suivantes ont été imposées lors de cette génération :

- le test de vacuité `ndfs` (cf. stratégie 4) sur l'automate du produit $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$ doit prendre entre quinze secondes et trente minutes. Pour les formules violées (i.e, produisant un contre-exemple) il doit exister un ordre de visite respectant cette contrainte ;
- pour chaque modèle on souhaite avoir au minimum deux heures de calcul pour traiter toutes les formules telles que $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}} = \emptyset$ et au minimum deux heures de calcul pour les formules telles que $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}} \neq \emptyset$.

Modèle	Nb. formules		Nb états			Nb transitions			Nb SCC		
	Vérifiées	Violées	min	avg	max	min	avg	max	min	avg	max
adding.4	201	212	1	5	18	2	14	99	1	4	18
bridge.3	196	233	1	12	90	3	83	1 003	1	8	74
brp.4	26	34	2	9	33	4	48	363	1	6	24
collision.4	14	13	1	12	33	3	68	277	1	9	30
*cyclic-scheduler.*3	168	188	2	15	61	4	105	804	1	10	48
elevator.4	241	280	2	10	56	4	48	586	1	8	37
elevator2.3	30	41	1	6	17	2	17	99	1	5	16
exit.3	157	149	1	10	72	2	47	739	1	7	56
leader-election.3	323	310	1	27	143	34	248	1 609	1	17	123
production-cell.3	214	238	2	14	108	7	88	1 673	1	10	77

Table 6.2 – Informations sur les automates issus de la traduction des formules LTL.

Ces deux règles assurent que l'automate du produit synchronisé est complexe et stimulera tous les aspects tests de vacuité. La table 6.2 résume les informations collectées sur ces formules. En raison de la deuxième contrainte imposée, on remarque que le nombre de formules par modèle est très variable. Ainsi les modèles ayant un espace d'état très grand ont généralement peu de formules associées : cela est dû au produit synchronisé qui est alors plus complexe et prend plus de temps à vérifier. L'étude de ce tableau montre que :

- le nombre de transitions maximum pour $\mathcal{A}_{\neg\varphi}$ est très variable puisque pour le modèle *production-cell.3* le plus gros automate possède dix-sept fois plus de transitions que le plus gros du modèle *adding.4*. Ce nombre est important car un automate de la formule avec peu d'états et beaucoup de transitions a plus de chance de se synchroniser avec l'automate du modèle ;
- le nombre d'états maximum pour $\mathcal{A}_{\neg\varphi}$ est lui aussi très variable : il y a un facteur six entre l'automate ayant le plus d'états pour le modèle *production-cell.3* et celui pour le modèle *elevator.2*. Ce nombre permet d'évaluer le nombre d'états de l'automate du produit puisque l'on a (en ne considérant que les états accessibles) : $|\mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\neg\varphi}| \leq |\mathcal{A}_{\mathcal{K}}| \times |\mathcal{A}_{\neg\varphi}|$;
- le nombre maximum de composantes fortement connexes par automate est lui aussi très variable puisque l'on observe dans le pire cas un facteur sept entre l'automate en ayant le moins et celui en ayant le plus. Ce nombre est important car plus il y a de composantes fortement connexes dans $\mathcal{A}_{\neg\varphi}$, plus il y de chances que l'automate du produit en ait puisque l'on a (en ne considérants que les états accessibles) : $SCC(\mathcal{A}_{\mathcal{K}} \otimes \mathcal{A}_{\neg\varphi}) \leq SCC(\mathcal{A}_{\mathcal{K}}) \times SCC(\mathcal{A}_{\neg\varphi})$;

Malgré ces différences pour les valeurs extrêmes, on remarque une certaine homogénéité sur les valeurs moyennes. En effet, les nombres moyens d'états, de transitions et de composantes fortement connexes sont relativement proches. Par exemple, si l'on regarde le nombre de composantes fortement connexes, on observe seulement des différences de l'ordre d'un facteur trois.

Note : dans ce jeu de tests, le plus petit automate est composé d'un état et deux transitions : il s'agit du plus petit automate de Büchi fort possible. Cet automate est issu de la formule LTL : $\mathsf{G}\,\mathsf{F}\,a$ et est représenté figure 6.1.

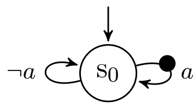


Figure 6.1 – tgba la formule ltl $\mathsf{G}\,\mathsf{F}\,a$ avec $\mathcal{F} = \{\bullet\}$.

6.1.3 Analyse du produit synchronisé

Les tables 6.3 et 6.4 présentent les principales mesures que nous avons effectuées sur les produits synchronisés en distinguant les cas où les automates ont un langage vide de ceux où ils ne l'ont pas.

On observe tout d'abord le phénomène d'explosion combinatoire évoqué tout au long de ce manuscrit. Pour le modèle *brp.4* par exemple, un des produits synchronisés a presque dix fois plus d'états que le modèle d'origine : on est néanmoins loin du pire cas (fois trente trois) attendu. On peut aussi noter que les automates du produit qui sont non vides ont tendance à avoir plus d'états que ceux qui sont vides. La construction de l'automate « à la volée » prend alors tout son sens puisque la détection prématurée d'un cycle acceptant ne requiert pas nécessairement la visite et le stockage de l'intégralité des états. De même, lorsqu'il n'y a pas de cycles acceptants seule la partie « utile » est construite.

Si la consommation mémoire des tests de vacuité est en moyenne proportionnelle au nombre d'états, leur temps d'exécution est proportionnel au nombre de transitions. On constate que le nombre moyen de transitions de $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$ est généralement deux fois supérieur à celui de $\mathcal{A}_{\mathcal{K}}$. Cependant de réelles différences existent : pour le modèle *leader-election.3*, il y a un produit (non vide) qui possède cinquante six fois plus de transitions que l'automate du modèle. Ce nombre important de transitions montre la nécessité de réduire au maximum le nombre de visites de chaque transition [2]. Ainsi, l'optimisation du `ndfs` qui vise a marquer prématurément des états comme rouge (optimisation de Gaiser et Schwoon [31], évoquée page 58) joue ce rôle pour les tests de vacuité basés sur un `ndfs`. Les tests de vacuité basés sur le calcul des composantes fortement connexes n'ont pas ce problème puisque chaque transition n'est visitée qu'une seule fois.

L'étude du nombre de composantes fortement connexes permet de voir l'effet du produit synchronisé sur les composantes de l'automate du modèle. On remarque tout d'abord que le nombre de composantes minimum de $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$ est très proche de celui de $\mathcal{A}_{\mathcal{K}}$. Cela signifie que lors du produit synchronisé un grand nombre de composantes de $\mathcal{A}_{\mathcal{K}}$ arrivent à se synchroniser avec celles de $\mathcal{A}_{\neg\varphi}$. En revanche, on remarque que le nombre de composantes maximum tend à exploser : par exemple, pour *brp.4* il existe un produit qui possède quarante sept fois plus de composantes fortement connexes que $\mathcal{A}_{\mathcal{K}}$. Ce nombre est important pour les tests de vacuité basés sur le calcul des composantes fortement connexes qui doivent maintenir les marques d'acceptation pour chaque composante.

De plus, on peut constater que le nombre d'états transients est réellement important puisqu'il y a en moyenne 99% des composantes fortement connexes qui sont transientes. Ce nombre doit aussi être mis en relation avec le nombre d'états. Ainsi, en moyenne 80% des états du produit sont transients.

2. La génération des successeurs peut être coûteuse et devient alors un critère déterminant lors du passage à l'échelle [40].

	Modèle	Nb Etats			Nb Transitions			Nb SCCs		
		min	avg	max	min	avg	max	min	avg	max
$\mathscr{L}(\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}) = \emptyset$	*adding.4*	3 370 654	5 637 711	24 772 713	5 683 941	10 725 851	61 976 700	3 370 654	5 635 309	24 772 713
	bridge.3	417 446	1 702 938	7 549 027	953 672	4 740 247	30 055 716	417 446	1 701 048	7 333 651
	brp.4	12 068 447	15 630 523	27 442 387	25 137 148	33 580 776	64 114 792	1 696 242	4 674 238	17 070 182
	collision.4	950 345	30 384 332	44 071 003	2 603 808	82 372 580	121 264 638	75 477	347 535	2 698 408
	cyclic-scheduler	175 827	724 400	2 832 020	1 239 613	6 274 289	37 208 333	16 383	453 547	2 563 651
	elevator.4	888 053	2 371 413	8 906 731	2 108 523	7 001 559	29 861 168	6	1 327 005	7 768 896
	elevator2.3	1 212 338	10 339 003	20 127 016	7 867 402	79 636 749	192 859 498	1	2 926 881	12 459 305
	exit.3	1 272 414	3 664 436	18 234 629	1 956 570	11 995 418	66 873 060	1 272 414	3 659 550	18 009 837
	leader-election.3	101 360	546 145	1 544 539	446 025	3 200 607	12 683 622	101 360	546 145	1 544 536
	production-cell.3	356 016	2 169 112	8 467 281	980 178	7 303 450	43 086 881	2	1 236 881	7 644 670
$\mathscr{L}(\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}) \neq \emptyset$	*adding.4*	3 370 706	7 720 939	21 015 503	5 684 045	14 341 202	45 604 091	3 370 706	7 716 385	21 015 503
	bridge.3	553 956	3 114 566	9 799 963	1 279 563	8 615 971	38 366 114	553 956	3 106 797	9 230 026
	brp.4	12 073 227	38 474 669	135 101 543	25 142 388	94 561 556	369 390 690	1 700 082	16 520 165	83 235 416
	collision.4	42 339 793	101 596 324	211 865 184	115 021 420	349 949 837	820 030 422	165 111	22 677 968	119 461 898
	cyclic-scheduler	237 594	1 364 512	4 439 774	1 656 907	12 368 800	47 472 225	16 413	711 794	3 175 259
	elevator.4	874 497	3 270 061	14 114 157	2 201 720	9 817 617	40 607 285	12	1 502 808	10 756 488
	elevator2.3	614 370	13 818 813	52 820 846	3 702 390	120 821 886	486 037 474	1	6 413 279	32 159 103
	exit.3	2 249 133	8 617 173	34 194 271	5 950 725	29 408 340	144 912 155	2 249 133	8 609 674	34 137 232
	leader-election.3	202 028	762 684	3 937 450	811 440	4 033 362	25 485 161	202 028	762 684	3 937 450
	production-cell.3	756 187	3 908 715	20 753 044	2 113 606	13 470 569	85 818 694	188 309	1 925 909	14 914 698

Table 6.3 – Informations en termes d'états, de transitions et de composantes fortement connexes du produit synchronisé du jeu de test.

	Modèle	Nb états transients			Taille max. DFS			Nb SCCs acceptantes			Taille SCCs	
		min	avg	max	min	avg	max	min	avg	max	min	max
$\mathscr{L}(\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}) = \emptyset$	*adding.4*	2 887 943	5 049 120	23 807 289	91	91	94	0	0	0	1	2
	bridge.3	377 454	1 528 310	7 103 557	46	86	123	0	0	0	1	3
	brp.4	1 642 503	4 616 488	17 016 443	116 801	117 533	135 739	0	0	0	1	4
	collision.4	75 460	346 946	2 698 375	25 833	927 659	1 205 256	0	0	0	1	10 252 826
	cyclic-scheduler	16 382	453 546	2 563 650	51	88 489	179 282	0	0	0	1	51 506 450
	elevator.4	5	1 327 003	7 768 891	16 328	252 692	513 630	0	0	0	1	5 286 856
	elevator2.3	0	2 926 880	12 459 304	47	993 479	1 027 758	0	0	0	1	5 286 856
	exit.3	1 228 960	3 105 503	17 542 085	32	33	37	0	0	0	1	5 286 856
	leader-election.3	101 359	546 143	1 544 532	56	142	156	0	0	0	1	7 233 308
	production-cell.3	1	1 236 879	7 644 669	17 169	148 268	228 982	0	0	0	1	1 454 044
$\mathscr{L}(\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}) \neq \emptyset$	*adding*	2 887 994	6 800 436	19 567 368	91	93	163	1	243 447	965 424	1	2
	bridge.3	476 928	2 773 490	9 018 583	47	87	125	16	78 253	278 662	1	4
	brp.4	1 646 323	16 398 743	83 076 699	115 997	133 072	398 547	1	23 243	106 214	1	10 252 826
	collision.4	165 061	22 616 630	119 455 718	732 252	1 192 221	1 504 028	9	2 841	15 768	1	21 742 200
	cyclic-scheduler	16 412	711 791	3 175 255	85 787	102 751	369 355	1	1	3	1	5 286 856
	elevator.4	9	1 502 803	10 756 481	98 267	258 906	490 157	1	1	20	1	6 463 784
	elevator2.3	0	6 413 272	32 159 063	45	827 727	2 055 769	1	5	27	1	3 161 396
	exit.3	1 922 439	7 237 286	30 637 150	33	33	36	108	395 455	1 502 325	1	5 286 856
	leader-election.3	202 027	762 680	3 937 444	151	151	160	1	1	4	1	669 663
	production-cell.3	188 306	1 925 905	14 914 695	19 340	167 007	607 030	1	1	4	1	7 667 712

Table 6.4 – Informations en termes d'états, de transitions et de composantes fortement connexes du produit synchronisé du jeu de test(suite).

Notons néanmoins que les modèles de type (a) et (c) ont tendance a avoir nettement moins d'états transients : leur structure complexe favorise l'apparition de nombreuses composantes fortement connexes.

Enfin, pour les produits non vides il est intéressant d'observer le nombre de composantes acceptantes. On constate ainsi que 0,05% des composantes sont acceptantes et que les produits synchronisés avec le modèle *cyclic-scheduler* possèdent au maximum trois composantes fortement connexes acceptantes. Ce nombre est très petit et montre donc la difficulté d'extraire les composantes acceptantes dans ce jeu de test (avec les contraintes que nous avons fixées).

L'étude de la structure des automates du produit a montré que :

- les produits synchronisés ont des structures complexes et sont souvent composés de composantes fortement connexes de grosse taille ;
- le nombre d'états transients est très important puisque ces deniers représentent l'immense majorité des composantes fortement connexes ;
- les composantes acceptantes représentent une infime partie des composantes de $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$;
- les espaces d'états des produits synchronisés ont des structures très variables. Ainsi, ils vont pouvoir stimuler les différents tests de vacuité notamment sur les aspects de gestion de la mémoire et de l'efficacité de détection des contre-exemples.

6.2 Analyse et performances

Cette section vise à évaluer les performances des tests de vacuité mentionnés tout au long de cette partie. Nous ferons ensuite lien entre ces performances et les structures des espaces d'états des automates du produit synchronisé.

6.2.1 Évaluation des tests de vacuité

Intéressons-nous maintenant à l'évaluation des tests de vacuité présentés table 6.5 sur le jeu de test de la section précédente. Tous ces algorithmes utilisent un DFS pour parcourir l'automate du produit synchronisé. Afin de pouvoir comparer ces différents algorithmes nous imposons que l'ordre de parcours de ce DFS sur le produit synchronisé soit le même pour tous les tests de

Abréviation	Algorithme	Page	Optimisations	Basé sur l'algorithme de ...
`ndfs`	Stratégie 4	57	—	—
`tec`	Stratégie 5	65	—	Tarjan
`tec+cs`	Stratégie 5	65	Pile compressée	Tarjan
`dec`	Stratégie 6	71	—	Dijkstra
`dec+cs`	Stratégie 6	71	Pile compressée	Dijkstra
`tuf`	Stratégie 7	82	—	Tarjan
`tuf+cs`	Stratégie 7	82	Pile compressée	Tarjan
`duf`	Stratégie 8	84	—	Dijkstra
`duf+cs`	Stratégie 8	84	Pile compressée	Dijkstra

Table 6.5 – Résumé des tests de vacuité évalués ainsi que leurs optimisations.

vacuité[3]. Les détails d'implémentation et les conditions d'expérimentations sont spécifiés en annexe A.

Le premier algorithme mentionné par la table 6.5 est le `ndfs` qui constitue à ce jour le test de vacuité le plus utilisé pour le *model checking* explicite. Nous le considérerons donc comme l'algorithme de référence tout au long de ce chapitre. Tous les autres tests de vacuité ont été présentés dans les chapitres 4 et 5 et peuvent être déclinés avec ou sans l'optimisation de la pile compressée présentée section 4.4.

6.2.1.1 Impact de la pile compressée

La table 6.6 montre la réduction mémoire induite par l'utilisation d'une pile compressée. Comme tous les tests de vacuité qui dérivent de l'algorithme de Tarjan (resp. Dijkstra) ont la même gestion de la pile nous ne les distinguons donc ici que par l'algorithme de calcul de composantes fortement connexes sur lequel ils sont basés.

	Pic observé pour la pile		Pic cumulé observé pour la pile	
	non-compressée	compressée	non-compressée	compressée
Dijkstra	965	68	515 673	18 565
Tarjan	1 205 265	737 789	310 868 872	23 236 2644

TABLE 6.6 – Impact de la pile compressée sur l'ensemble du jeu de tests.

Cette table présente donc aussi bien le pic maximum observé (avec ou sans compression) que le pic cumulé observé lors de l'exploration des différents produits synchronisés résultant du jeu de test décrit en section précédente. Le pic observé est considérablement plus faible pour les tests de vacuité basés sur l'algorithme de Dijkstra qu'il ne l'est pour ceux basés sur l'algorithme de Tarjan. Cet écart vient des différences de mise à jour de l'information que deux états sont dans la même composante fortement connexe :

- dans l'algorithme de Tarjan, cette information est maintenue pour chaque état jusqu'à ce qu'il soit dépilé de la pile DFS ;
- dans l'algorithme de Dijkstra, dès que deux états sont détectés comme étant dans la même composante la pile est réduite au maximum ;

Ainsi, la pile des lowlinks de Tarjan suit exactement la même évolution que la pile DFS. Si l'on regarde la table 6.4, on constate en effet qu'il existe un produit (dans le modèle *collision.4*) pour lequel la taille maximum de la pile DFS correspond exactement au pic observé ici. C'est cette différence de gestion de la pile qui permet aux algorithmes basés sur Dijkstra d'avoir un pic qui est mille fois plus petit que ceux basés sur Tarjan. On peut constater que la mise en place de la pile compressée permet encore de réduire l'empreinte mémoire puisque :

- pour l'algorithme de Dijkstra la pile compressée est quatorze fois plus petite que l'algorithme original ;
- pour l'algorithme de Tarjan la pile compressée est presque deux fois plus petite que la pile originale.

3. À l'exception du `ndfs` qui nécessite une dégénéralisation lorsque le nombre de marques d'acceptation est strictement supérieur à un. Dans ce cas, l'automate du produit synchronisé est différent. Néanmoins, les transitions du modèles sont toujours sélectionnées dans le même ordre ce qui permet une comparaison honnête.

Pour les tests de vacuité chaque élément de la pile maintient deux entiers (une position et un ensemble de marques d'acceptation). Sur l'ensemble du jeu de test le gain de la pile compressée est de 994 216 entiers pour les tests de vacuité basés sur l'algorithme Dijkstra et de 157 012 456 entiers pour ceux basés sur l'algorithme Tarjan. Cette étude montre que l'idée de fusionner les informations liées aux états transients (ou qui sont encore considérés comme tels par l'algorithme) permet un gain mémoire non négligeable.

6.2.2 Performances des tests de vacuité

Maintenant que nous avons vu comment réduire la consommation mémoire des tests de vacuité basés sur le calcul des composantes fortement connexes, intéressons-nous à leurs performances.

La figure 6.2 présente pour chaque test ses performances par rapport au `ndfs` sur l'ensemble du jeu de test. Pour ces diagrammes, lorsqu'un point est en dessous de la diagonale, cela signifie que le `ndfs` est moins rapide. Nous constatons tout d'abord que ces tests de vacuité semblent avoir des performances comparables sur les produits vides et que les différences n'affectent que les produits non vides. Ces différences sont dues à :

- *une détection au plus tôt* pour les tests de vacuité basés sur l'algorithme de Dijkstra : dès que toutes les transitions formant un cycle acceptant ont été visitées ce dernier est détecté ;
- *une recherche de cycle acceptant répétée* : dès qu'une transition acceptante est dépilée une recherche de cycle acceptant est lancée pour les NDFS ;
- *une détection retardée* : les marques d'acceptation sont progressivement remontées à la racine de la composante fortement connexes lorsque les états sont dépilés de la pile DFS pour les tests de vacuité basés sur l'algorithme de Tarjan.

Ainsi les scatterplots de la figure 6.2 montrent que les algorithmes `tec` et `tuf`, qui sont basés sur l'algorithme de Tarjan, ont de mauvaises performances sur un certain nombre de produits non vides. Les algorithmes `dec` et `duf` offrent quant à eux de meilleures performances sur ces produits grâce à leur capacité a détecter un cycle acceptant rapidement.

Ces résultats sur quelques formules isolées ne doivent néanmoins pas masquer ceux sur le jeu de test complet. La figure 6.7 présente le temps cumulé nécessaire à chaque algorithme pour vérifier l'ensemble des produits vides et non vides.

On constate ainsi un écart de 15% (sur l'ensemble du jeu de test) entre le test de vacuité le plus lent (`ndfs`) et le plus rapide (`duc`). De plus, tous les tests de vacuité basés sur un calcul des composantes fortement connexes sont comparables puisque l'écart maximum est de 5% (entre `duc` et `tec+cs`). On remarque aussi que l'utilisation d'une pile compressée n'implique qu'un surcoût de 1% ce qui semble acceptable au vu des gains mémoire observées dans la section précédente. Enfin, l'utilisation d'une structure d'union-find permet un gain d'environ 3% sur l'ensemble de ce jeu de test. Cet écart est dû au transfert retardé des états vers l'ensemble stockant les états morts.

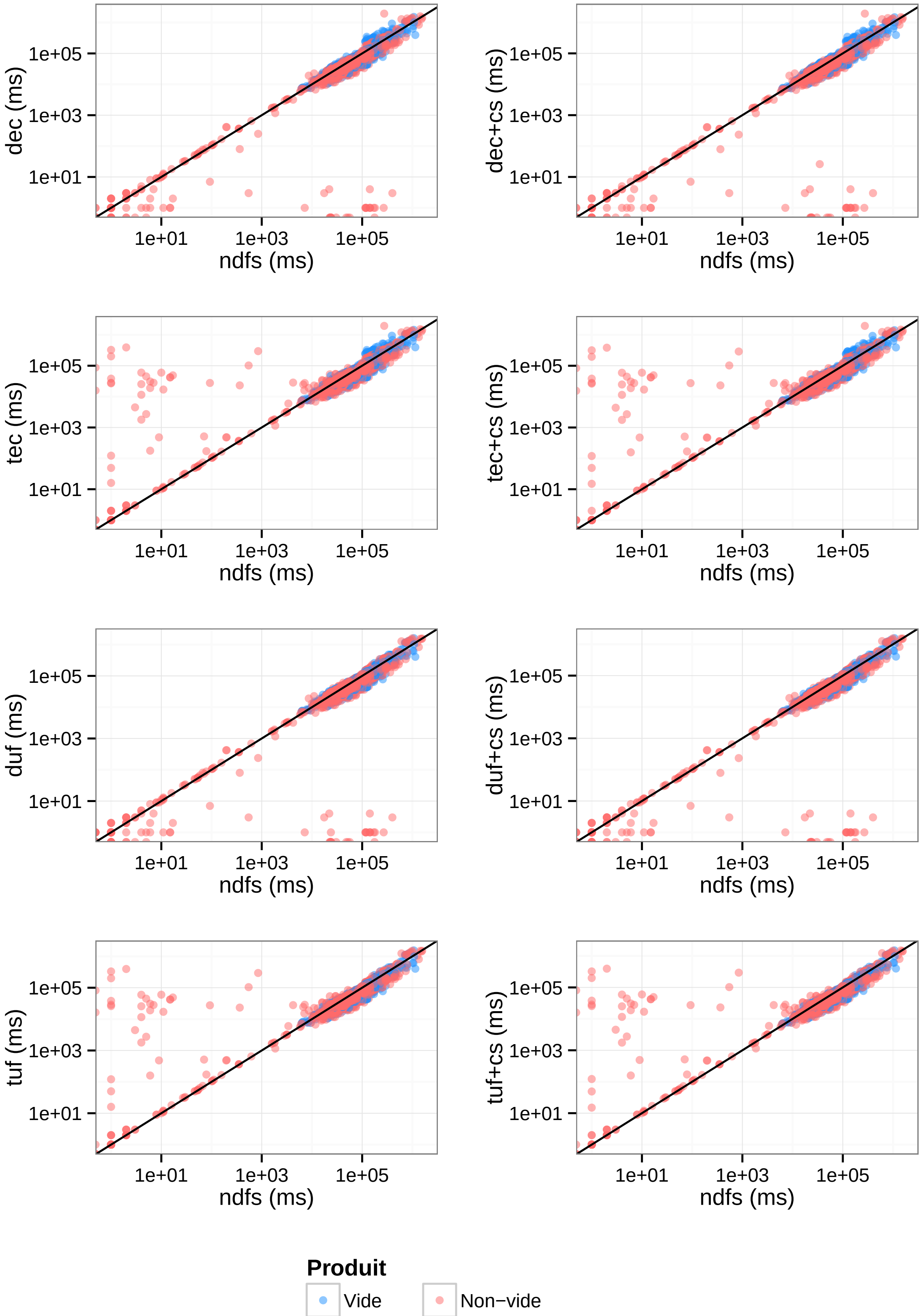


Figure 6.2 – Temps d'exécutions des tests de vacuité sur l'ensemble du jeu de tests comparés au `ndfs`.

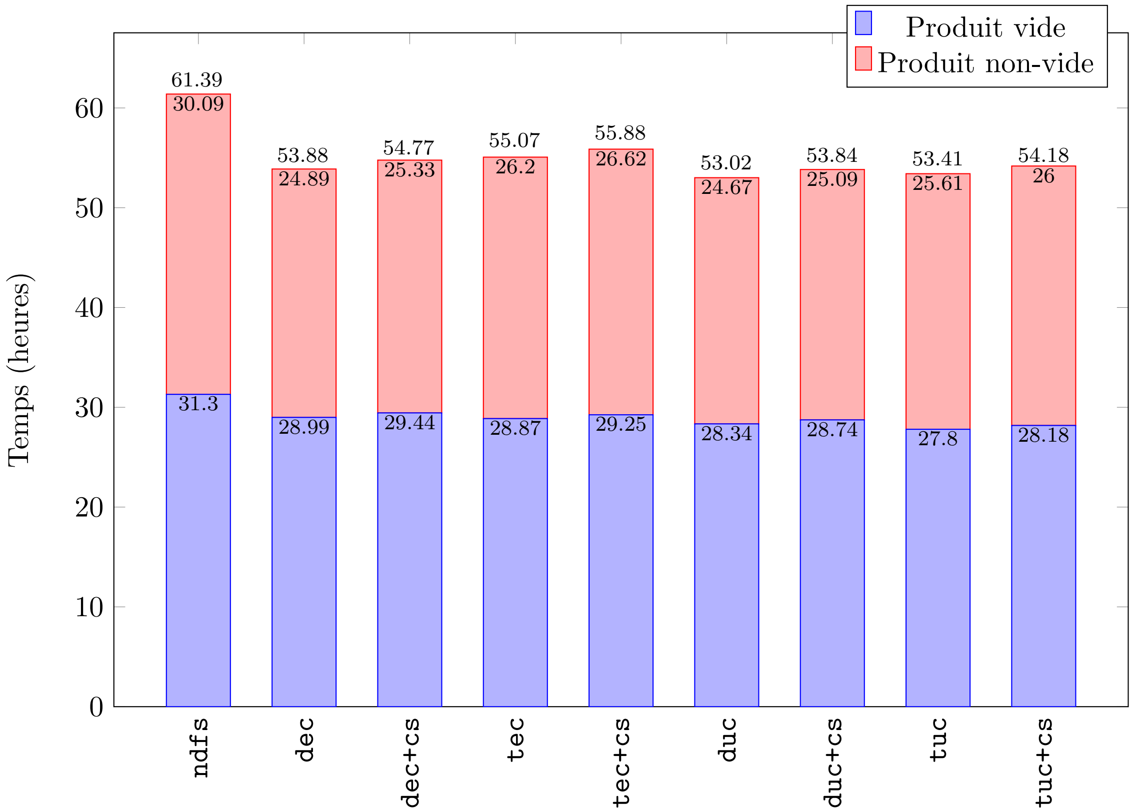


Figure 6.3 – Temps d'exécution cumulé sur l'ensemble du jeu de test pour les formules vérifiées (en bas) et violées (en haut).

Les mauvaises performances du `ndfs` sont essentiellement dues aux cas où une dégénéralisation est requise. La table 6.7 montre l'impact de cette opération sur les temps des tests de vacuité. On remarque ainsi que le `ndfs` offre de bonnes performances lorsqu'il n'y a qu'une seule marque d'acceptation. En revanche, plus le nombre de marques d'acceptation augmente, plus les performances se dégradent. Pour les cas où la formule possède cinq marques d'acceptation le `ndfs` met en moyenne 40% de temps de plus que tous les autres algorithmes. Ainsi l'augmentation du nombre de marques d'acceptation pénalise sérieusement le `ndfs` qui doit manipuler un

Algorithme	Nombre de marques d'acceptation				
	1	2	3	4	5
`ndfs`	100%	100%	100%	100%	100%
`tec`	101%	75%	71%	64%	58%
`dec`	101%	78%	72%	67%	60%
`tuf`	105%	76%	72%	68%	61%
`duf`	105%	74%	72%	65%	60%

Table 6.7 – Impact en temps de la dégénéralisation sur les tests de vacuité ; `ndfs` constitue la référence.

automate de la formule ayant plus d'états et de transitions. Lors de la vérification sous hypothèses d'équité, le nombre de marque d'acceptation augmente rapidement. Cette table montre l'importance d'utiliser des tests de vacuité supportant naturellement des conditions d'acceptation généralisées.

6.3 Conclusion

Dans ce chapitre, nous avons étudié les performances des tests de vacuité sur des automates forts. Pour cela nous avons élaboré un nouveau jeu de test basé sur certains modèles présents dans BEEM. Les contraintes imposées lors de la génération des formules associées nous assurent que la vérification de ces formules ne sera pas triviale.

L'étude de ce jeu de test a montré que les produits synchronisés ont des structures complexes et sont majoritairement composés de composantes fortement connexes transientes. Ces états ne sont généralement pas distingués des autres et les tests de vacuité basés sur le calcul des composantes fortement connexes doivent maintenir des informations pour chaque état ce qui peut être coûteux. Nous avons montré en section 4.4 que l'utilisation d'une pile compressée pour stocker l'information liée à ces états était possible. L'évaluation de cette technique sur notre jeu de test a montré des gains mémoires significatifs avec un surcoût temporel négligeable.

Nous avons ensuite montré que les performances du `ndfs` étaient largement impactées par la dégénéralisation. Cette dégénéralisation peut être évitée en utilisant des tests de vacuité généralisés. Tous les tests généralisés que nous avons présentés ont des performances comparables. Néanmoins, nous avons vu non seulement que l'utilisation d'une structure d'union-find permet un léger gain, mais aussi que les tests de vacuité basés sur l'algorithme de Dijkstra sont légèrement plus performants que ceux basés sur l'algorithme de Tarjan pour les produits non vides. Pour les produits vides nous avons constaté l'inverse. Ces deux familles d'algorithmes sont donc utiles : en phase de conception (i.e. lorsque les erreurs sont nombreuses) les tests de vacuité basés sur l'algorithme de Dijkstra doivent être privilégiés, tandis en phase de mise au point ceux basés sur l'algorithme de Tarjan peuvent l'être.

Troisième partie

Contributions aux tests de vacuité parallèles

Chapitre 7

Mieux exploiter les forces de l'automate de la propriété

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

Brian Kernighan

Sommaire

Les tests de vacuité de la partie I permettent de traiter efficacement les automates issus de la synchronisation entre une structure de Kripke et un automate fort (détails chapitres 3, 4, et 5). Un automate est dit fort dès qu'une de ses composantes fortement connexes possède des cycles acceptants et des cycles non-acceptants. L'étude de ces cycles permet de caractériser chacune des composantes fortement connexe pour déduire la force de l'automate.

Néanmoins plusieurs composantes fortement connexes ayant des caractéristiques différentes peuvent cohabiter dans le même automate : dans ce cas il est dit multi-forces. *Ce type automates reste relativement peu étudié : ce chapitre s'intéresse d'abord à classer finement les composantes fortement connexes de l'automate de la propriété puis montre comment l'approche par automates pour le* model checking *peut en tirer parti.*

7.1 Détection des forces des composantes fortement connexe

L'idée de classer les composantes fortement connexes en fonction de leur force a été suggérée par Edelkamp et al. [26] pour optimiser les tests de vacuité basés sur un NDFS lorsque l'automate de la propriété est multi-forces (cf. section 7.2). Cependant, cette classification n'intègre pas la notion de composante fortement connexe complète qui permet pourtant de simplifier les tests de vacuité (cf. section 3.4). Dans cette section, nous raffinons cette classification pour l'adapter aux automates généralisés et y intégrer la notion de composante fortement connexe complète.

Définition 19 – Force d'une composante fortement connexe

Une composante fortement connexe est dite :

Non-acceptante	: si la composante ne contient aucun cycle acceptant ;
Intrinsèquement terminale	: si tous les cycles de la composante sont acceptants et la composante fortement connexe est complète (détails définition 17) ;
Intrinsèquement faible	: si tous les cycles de la composante sont acceptants et la composante fortement connexe n'est pas intrinsèquement terminale (la composante n'est pas complète) ;
Forte	: si la composante possède de cycles acceptants et des cycles non acceptants.

La classification proposée à la définition 19 définit une partition des composantes fortement connexes de l'automate et la figure 7.1 présente un exemple d'automate dans lequel chaque force est représentée. L'unique composante non acceptante est la composante C_3. La composante C_1 est forte car elle est acceptante mais il est possible d'avoir un cycle non-acceptant qui ne visite que s_1. La composante C_2 est intrinsèquement faible car tous ses cycles sont acceptants. Enfin, la composante C_4 est intrinsèquement terminale car tous les cycles sont acceptant et la composante est complète (puisque l'unique transition est étiquetée par $\top$).

Note : les composantes fortement connexes non acceptantes *inutiles*, i.e. il n'existe pas de chemin depuis celles-ci vers une composante acceptante, ne sont pas distinguées dans cette caractérisation puisque non pertinentes pour les tests de vacuité [1].

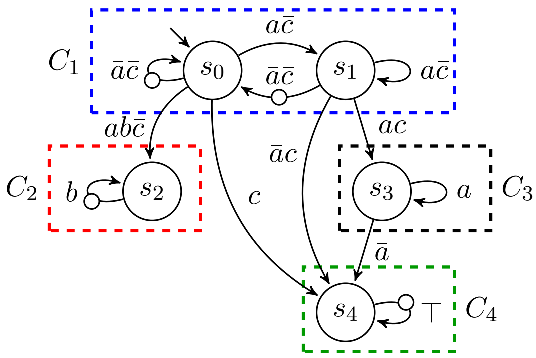


Figure 7.1 – TGBA $\mathcal{A}_{\neg\varphi}$ pour $\varphi = \neg((\mathsf{G}\, a \rightarrow \mathsf{G}\, b)\, \mathsf{W}\, c)$.

Cette classification des composantes fortement connexes permet de généraliser la définition de force d'un automate donnée à la page 49 (définition 18). La définition 20 présente cette généralisation.

1. Dans la pratique les outils de traduction de formules LTL ne génèrent pas de composantes inutiles.

Définition 20 – Force d'un automate

Un automate est dit :

Intrinsèquement terminal : si toutes ses composantes fortement connexes acceptantes sont intrinsèquement terminales ;

Intrinsèquement faible : si toutes ses composantes fortement connexes acceptantes sont intrinsèquement faibles ou intrinsèquement terminales ;

Fort : si toutes ses composantes fortement connexes sont fortes ;

Multi-forces : sinon.

Comme il ne s'agit que d'une généralisation, les tests de vacuité pour les automates terminaux, faibles et forts sont respectivement applicables aux automates intrinsèquement terminaux, intrinsèquement faibles et forts. Enfin, si un automate multi-forces contient une composante fortement connexe forte, un test de vacuité fort est requis, sinon un test de vacuité faible suffit. L'automate de la figure 7.1 est donc multi-forces et requiert un test de vacuité fort.

La figure 7.2 montre l'inclusion de ces forces d'automates. On voit ainsi que toutes les classes « intrinsèques » sont plus expressives que les classes généralement utilisées et qu'elles forment une hiérarchie tandis que les automates forts constituent une classe à part.

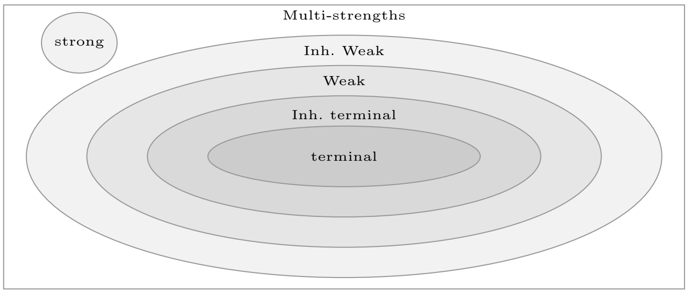


Figure 7.2 – Classification des automates de Büchi.

Pour déterminer le test de vacuité à utiliser sur l'automate du produit synchronisé, il faut d'abord calculer les composantes fortement connexes de l'automate de la propriété puis regarder les marques d'acceptation associées. Cela peut être fait simplement en adaptant les algorithmes de Tarjan ou Dijkstra comme montré dans les sections précédentes. Si une composante n'est pas acceptante (i.e. toutes les marques d'acceptation ne sont pas présentes) sa classification est trouvée. Sinon, plusieurs heuristiques peuvent être mises en place pour détecter sa force.

Heuristique syntaxique Lors de la traduction d'une formule LTL en un automate, l'algorithme proposé par Couvreur [19] étiquette les états de l'automate de la propriété par des formules LTL. Celles-ci représentent la formule vérifiée depuis chaque état.

Exemple.
Dans la figure 7.1 une telle traduction effectue l'association suivante :

$$
\begin{array}{llll}
s_0 : & (\mathsf{G}\, a \rightarrow \mathsf{G}\, b) \,\mathsf{W}\, c & s_1 : & \mathsf{F}\, \neg a \wedge ((\mathsf{G}\, a \rightarrow \mathsf{G}\, b) \,\mathsf{W}\, c) \\
s_2 : & \mathsf{G}\, b & s_3 : & \mathsf{F}\, \neg a \\
s_4 : & \top & &
\end{array}
$$

Cette étiquette, combinée à la classification de Černá et Pelánek [15], est ensuite utilisée pour détecter la force d'une composante fortement connexe. Ainsi, les états s_0 et s_1 sont syntaxiquement associés à des formules de récurrence, donc la composante à laquelle ils appartiennent peut être considérée comme forte. Comme l'état s_2 est associé à une formule de sûreté, la composante C_2 peut être considérée comme intrinsèquement faible, tandis que la composante C_4 peut être considérée comme intrinsèquement terminale puisque s_4 est associé à une formule de garantie.

Heuristique structurelle L'heuristique précédente présente des contraintes puisqu'elle suppose que l'automate est issu de la traduction de formules LTL. Elle fait ainsi des hypothèses sur l'algorithme de traduction utilisé, et certaines composantes peuvent être surclassées à cause de formules pathologiques [2]. De plus, l'heuristique précédente n'est pas applicable lorsque l'automate est issu d'une dégénéralisation car l'étiquette est perdue lors de cette opération. Pour palier cela, la caractérisation de Bloem et al. [10] sur l'automate peut être raffinée au niveau des composantes fortement connexes. Ainsi une composante sera considérée comme intrinsèquement faible si toutes les transitions sont étiquetées par l'ensemble des éléments de $\mathcal{F}$; si elle est en plus complète elle sera considérée comme intrinsèquement terminale.

Heuristique intrinsèque L'heuristique précédente présente elle aussi des inconvénients : certaines composantes fortement connexes peuvent être mal classifiées. La figure 7.3 montre un exemple dans lequel la composante est considérée comme forte alors qu'elle est intrinsèquement faible : cela est dû à la la transition (q_1, q_0) qui ne porte pas la marque d'acceptation $\circ$. Une étude des cycles élémentaires (qui ne passent qu'une unique fois par chaque état) montre néanmoins que dans cette composante tous les cycles élémentaires sont acceptants. Cette heuristique propose donc d'énumérer l'ensemble des cycles élémentaires pour détecter si une composante est intrinsèquement faible. Dès qu'un cycle non acceptant est détectée, la composante peut être déclarée comme forte. Sinon, il n'existe pas de tels cycles et tester si la composante est complète permet de savoir si elle est faible ou terminale (elle est terminale si elle est en plus complète). Notons cependant que l'énumération des cycles élémentaires est coûteux puisqu'elle a une complexité exponentielle.

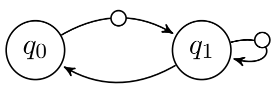


Figure 7.3 – SCC intrinsèquement faible mal classifiée par l'heuristique structurelle car la transition (q_1, q_0) n'est pas acceptante.

Nous avons effectué des tests [73] sur 10 000 formules aléatoires pour analyser l'efficacité de ces heuristiques. Ces formules ont toutes été traduites en TGBA par l'algorithme de Couvreur [19]

2. Une formule est dite pathologique si elle est reconnue d'une classe mais qu'elle peut être traduite en un automate d'une classe inférieure.

(afin de pouvoir évaluer l'heuristique syntaxique). L'heuristique intrinsèque étant théoriquement la plus forte nous considérons qu'elle classifie correctement l'ensemble des composantes fortement connexes. Les composantes intrinsèquement faibles sont quant à elles bien classées dans 99,85% des cas avec l'heuristique structurelle et dans 87,5% des cas avec l'heuristique syntaxique. Enfin les trois heuristiques classent correctement 100% des composantes intrinsèquement terminales. L'heuristique structurelle constitue donc un bon compromis car ces tests montrent qu'elle est seulement trois fois plus lente que l'heuristique syntaxique et dix fois plus rapide que l'heuristique intrinsèque. De plus, les 0,15% des cas où l'heuristique structurelle échoue sont essentiellement composés de formules pathologiques. Par la suite nous utiliserons donc l'heuristique structurelle pour trouver la classe associée à une composante fortement connexe.

Note : comme l'automate de la propriété est généralement très petit par rapport à l'automate du système, une analyse complète n'est pas coûteuse et permet en plus d'appliquer des opérations de réductions sur l'automate de la propriété [2]. L'inconvénient est que l'approche n'est pas complètement « à la volée » puisque seule la structure de Kripke est calculée dynamiquement [3].

7.2 Test de vacuité basé sur un NDFS avec forces

L'unique test de vacuité en tirant parti des automates multi-forces a été proposé par Edelkamp et al. [26] dans le cadre d'un algorithme basé sur un NDFS. L'idée est d'utiliser la force des composantes de l'automate de la propriété pour basculer dynamiquement entre les trois tests de vacuité présentés chapitre 3.

Lorsque la projection d'un état sur l'automate de la propriété appartient à une composante intrinsèquement faible ou terminale il n'est pas nécessaire de lancer un parcours imbriqué : la détection de cycles acceptants peut se faire simplement en détectant une transition dont la cible est déjà sur la pile DFS. Comme nous ne considérons ici que l'heuristique structurelle, l'optimisation de Schwoon et Esparza [77] suffit puisqu'une transition acceptante dont la cible est un état *Cyan* permet de trouver un contre-exemple. En effet, une transition acceptante allant sur un état cyan ferme nécessairement un cycle acceptant.

Lorsqu'un parcours imbriqué est nécessaire, Edelkamp et al. proposent de limiter ce parcours aux seuls états pouvant contenir un contre-exemple, i.e. ceux dont la projection est dans la même composante fortement connexe forte. Cette optimisation permet dans le meilleur des cas de diviser par deux le nombre de transitions visitées. La stratégie 9 présente les modifications à apporter à la stratégie 4 pour y intégrer cette modification. Seules les lignes 19, 20, 30, 31, 38 et 39 (en rouges, marquées d'une étoile) varient. Les lignes 19 et 20 d'une part et 30 et 31 d'autre part restreignent le déclenchement d'un parcours imbriqué aux composantes fortes de l'automate de la propriété. La ligne 38 teste si les projections sur $\mathcal{A}_{\neg\varphi}$ de s et $t.dst$ sont dans la même composante, tandis que la ligne 39 ignore les états pour lesquels ce n'est pas le cas.

Dans le cas où le système est non-bloquant, la méthode $\texttt{PUSH}_{\texttt{NDFS}}$ peut être modifiée pour tester si la projection du nouvel état sur $\mathcal{A}_{\neg\varphi}$ appartient à une composante fortement connexe intrinsèquement terminale. Si c'est le cas un contre-exemple est alors détecté.

3. L'approche « à la volée » suggère de générer l'automate de la propriété dynamiquement. Dans la pratique cela n'est pas fait pour pouvoir effectuer des réductions (en terme d'états et de transitions) sur cet automate et ainsi combattre l'explosion combinatoire.

```
Structures supplémentaires :
  enum color { Blue, Red, Cyan }
  struct Step {src : Q,     succ : 2^Δ,
               acc : 2^F,   allred : bool} // Refinement of Step of Algo. 1

Variables Locales supplémentaires :
  coloredset : map of Q ↦ ⟨c : color⟩

PUSH_NDFS(acc ∈ 2^F, q ∈ Q) → int
  coloredset.insert(⟨ q, Cyan ⟩)
  dfs.push(⟨ q, succ(q), acc, ⊤ ⟩)

GET_STATUS_NDFS(q ∈ Q) → Status
  if coloredset.contains(q) then
    return coloredset.get(q).c = Red ? DEAD : LIVE
  return UNKNOWN

UPDATE_NDFS(acc ∈ 2^F, dst ∈ Q)
  if acc = 2^F then
    if coloredset.get(dst).c = Cyan then
      report Accepting cycle detected!
    else
      if isStrongSCCof(P_{|A_¬φ}(dst), A_¬φ) then
        nested_dfs(dst)
      coloredset.get(dst).c ← Red
  else
    dfs.top().allred ← ⊥

POP_NDFS(s ∈ Step)
  dfs.pop()
  coloredset.get(s.src).c ← Blue
  if s.allred = ⊤ then
    coloredset.get(s.src).c ← Red
  else if s.acc = 2^F then
    if isStrongSCCof(P_{|A_¬φ}(s.src), A_¬φ) then
      nested_dfs(s.src)
    coloredset.get(s.src).c ← Red
  else
    if ¬ dfs.empty() then
      dfs.top().allred ← ⊥

// Also known as Red-DFS
forall the Transition t ∈ succ(q) do
  if ¬ sameSCCof(P_{|A_¬φ}(q),P_{|A_¬φ}(t.dst), A_¬φ) then
    continue
  if coloredset.get(t.dst).c = Cyan then
    report Accepting cycle detected!
  if coloredset.get(t.dst).c = Blue then
    coloredset.get(t.dst).c ← Red
    nested_dfs(t.dst)
```

Stratégie 9: Modification de l'algorithme de Gaiser et Schwoon [31] pour y intégrer l'optimisation de Edelkamp et al. [26] (suite).

Note : Cette optimisation présentée par Edelkamp et al. [26] sur les tests de vacuité non généralisés est applicable directement à tous les tests basés sur un NDFS, notamment ceux généralisés proposés par Couvreur et al. [20] ou par Tauriainen [84]. Les tests de vacuité basés sur le calcul des composantes fortement connexes traitent naturellement les composantes faibles.

7.3 Découpage de l'automate de la propriété

L'algorithme de la section précédente exploite donc au mieux les automates multi-forces pour réduire les parcours imbriqués et détecter au plus tôt les cycles acceptants pour les composantes faibles et terminales. Cependant il maintient les mêmes structures de données que l'algorithme original et effectue des tests supplémentaires pour adapter dynamiquement son comportement. L'idée proposée ici exploite aussi les automates multi-forces mais sans alourdir le test de vacuité.

Un automate multi-forces contient des parties qui sont fortes, faibles ou terminales pour lesquelles il existe des tests de vacuité dédiés. Plutôt que d'adapter le test de vacuité à l'automate, on peut adapter l'automate au test de vacuité. Ainsi pour un automate multi-forces donné, il suffit de construire un automate fort, un automate intrinsèquement faible et un automate intrinsèquement terminal pour bénéficier des meilleurs tests de vacuité.

Définition 21 – Automates dérivés

Soit T, W et S l'ensemble des transitions appartenant respectivement aux composantes intrinsèquement terminales, faibles et fortes. Pour un ensemble de transitions X donné, $Pre(X)$ représente l'ensemble des états pouvant atteindre une transition de X ; $Pre(X)$ contient toujours au minimum l'état initial de l'automate (même si X est vide).

Soit $\mathcal{A} = \langle\, Q,\, q_0,\, AP,\, \mathcal{F},\, \Delta\,\rangle$ un TGBA et $\bullet$ une marque d'acceptation. Les automates $\mathcal{A}_T = \langle\, Q_T,\, q_0,\, AP,\, \{\bullet\},\, \Delta_T\,\rangle$, $\mathcal{A}_W = \langle\, Q_W,\, q_0,\, AP,\, \{\bullet\},\, \Delta_W\,\rangle$, et $\mathcal{A}_S = \langle\, Q_S,\, q_0,\, AP,\, \mathcal{F},\, \Delta_S\,\rangle$, représentant respectivement les comportement acceptés par les composantes intrinsèquement terminales, intrinsèquement faibles et fortes peuvent être construits de la façon suivante :

$$\begin{cases} Q_T &= \mathrm{Pre}(T) \\ \Delta_T &= \{(s,\ell,c,d) \mid \exists\, t=(s,\ell,\_,d)\in\Delta \text{ avec } s,d\in Q_T \text{ et } \begin{cases} c=\{\bullet\} & \text{si } t\in T \\ c=\emptyset & \text{sinon}\end{cases} \} \end{cases}$$

$$\begin{cases} Q_W &= \mathrm{Pre}(W) \\ \Delta_W &= \{(s,\ell,c,d) \mid \exists\, t=(s,\ell,\_,d)\in\Delta \text{ avec } s,d\in Q_W \text{ et } \begin{cases} c=\{\bullet\} & \text{si } t\in W \\ c=\emptyset & \text{sinon}\end{cases} \} \end{cases}$$

$$\begin{cases} Q_S &= \mathrm{Pre}(S) \\ \Delta_S &= \{(s,\ell,c,d) \mid \exists\, t=(s,\ell,c,d)\in\Delta \text{ avec } s,d\in Q_S\} \end{cases}$$

Cette décomposition est telle que :

$$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}_T)\ \cup\ \mathscr{L}(\mathcal{A}_W)\ \cup\ \mathscr{L}(\mathcal{A}_S)$$

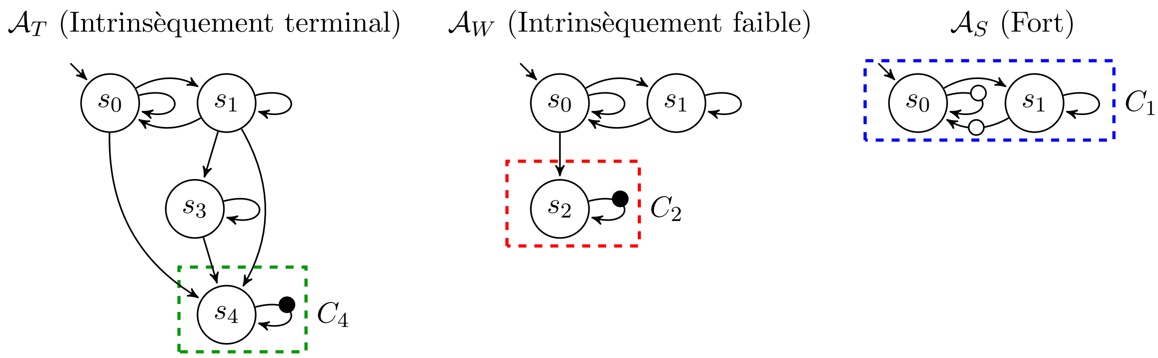


Figure 7.4 – Décomposition de l'automate de la Fig. 7.1 en trois automates.

Cette décomposition conserve le langage reconnu par $\mathcal{A}$: comme toute composante fortement connexe acceptante de $\mathcal{A}$ est présente dans au moins un des automates issus de la décomposition, n'importe quel chemin acceptant de $\mathcal{A}$ se retrouve tel quel dans un des trois automates.

Intuition de la preuve : ($\subseteq$) Un mot accepté par $\mathcal{A}$ est reconnu par une exécution qui va être capturée par une composante fortement connexe de $\mathcal{A}$. Comme toutes les composantes fortement connexes appartiennent à un des trois automates, cette composante fortement connexe est focément reproduite de manière acceptante dans un des trois automates dérivés, et elle est nécessairement accessible depuis l'état initial. ($\supseteq$) Comme les trois automates sont des restrictions de $\mathcal{A}$, un mot accepté par l'un d'entre eux est nécessairement accepté par $\mathcal{A}$.

Exemple.
La figure 7.4 présente la décomposition de l'automate de la figure 7.1 en appliquant la définition 21. Chacun des automates ne possède plus qu'une seule force de composante fortement connexe acceptante. Dans $\mathcal{A}_T$ (resp. $\mathcal{A}_T$) toutes les composantes non intrinsèquement terminales (resp. intrinsèquement faible) de l'automate original sont alors non acceptantes. Les composantes inutiles sont supprimées des automates ce qui permet de réduire leur taille : par exemple l'automate fort n'est plus composé que d'une unique composante fortement connexe.

Cette décomposition peut être exploitée pour améliorer le schéma de vérification présenté section 2. Une première idée consiste à choisir un ordre dans lequel on effectue les tests de vacuité, puis de les lancer de manière séquentielle, i.e. dès qu'un test de vacuité se termine et si aucun contre-exemple n'est détecté par le test de vacuité « suivant » est lancé. Cette idée suppose que les cycles acceptants se situent plus particulièrement dans l'une ou l'autre des composantes fortement connexes. Cependant les travaux de Geldenhuys et Valmari [35] montrent qu'il est difficile de trouver des heuristiques permettant d'orienter efficacement les tests de vacuité vers des composantes pouvant contenir des cycles acceptants : trouver le « bon » ordre dépend alors de l'automate que l'on vérifie. De plus, dans le pire cas l'intégralité du produit est construit pour les trois automates et une séquentialisation peut tripler le temps de calcul.

Une autre idée est de décomposer l'automate de la propriété en trois automates puis de lancer trois procédures de vérification en parallèle. Dès qu'un contre-exemple est trouvé les autres procédures s'arrêtent, sinon il faut attendre que le dernier test de vacuité ait terminé. Comme dans le pire cas l'automate issu de la décomposition est l'automate initial, le temps de calcul reste inchangé. Dans la pratique, l'automate issu de la décomposition est plus petit, et on peut espérer une réduction du temps de calcul. La figure 7.5 présente cette nouvelle approche.

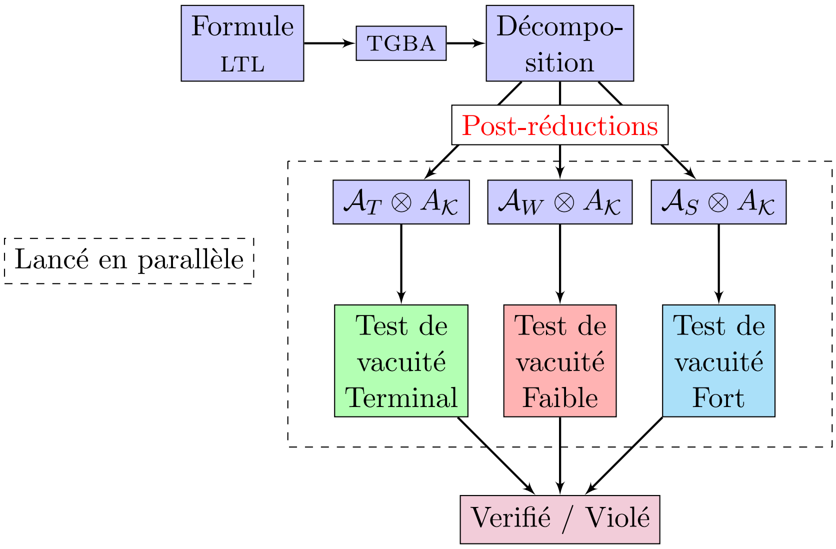


Figure 7.5 – Approche automate tirant parti des automates multi-forces.

Cette nouvelle approche est intéressante pour plusieurs raisons :

1. elle permet de paralléliser naturellement la vérification d'un propriété exprimée sous la forme d'un automate multi-forces,
2. elle assure que le test de vacuité utilisé pour chaque vérification est exactement celui nécessaire à la vérification de la propriété,
3. elle permet une combinaison avec d'autres techniques de vérifications telles que des approches symboliques,
4. des opérations de réductions sont possibles sur chaque automate après décomposition ce qui aide à combattre l'explosion combinatoire de la taille du produit,
5. ces réductions permettent de réduire le nombre de propositions observées ce qui est important pour certaines techniques basées sur de l'ordre partiel,
6. elle est compatible avec tous les tests de vacuité (même symboliques [73]) contrairement à l'approche proposée par Edelkamp et al. [26].

Perspectives Certains mots peuvent être reconnus par deux composantes fortement connexes de forces différentes. Dans ce cas, notre approche peut vérifier deux fois en parallèle les mêmes comportements. Pour palier cela une approche basée sur les automates non-ambigus [14] semble possible. Un automate non-ambigu est un automate dans lequel il n'existe qu'une seule façon de reconnaître un mot, i.e. ce mot ne sera reconnu que par une composante fortement connexe donné. Comme toute formule LTL peut être transformée en un automate non-ambigu [14], une telle approche permet de s'assurer que les langages des trois automates sont disjoints tout en conservant la propriété que leur union constitue le langage original. Ainsi chaque comportement n'est vérifié qu'une seule fois (à l'inverse de ce que peut faire notre approche). L'inconvénient principal est que les automates non-ambigu sont plus gros (en terme d'états et de transitions) que les TGBA : cela peut alors augmenter l'explosion combinatoire. L'analyse de l'impact des automates non-ambigus sur le *model checking* et plus particulièrement sur la décomposition proposée dans ce chapitre constituerait une étude intéressante.

7.4 Conclusion

Ce chapitre s'intéresse à une classe d'automates particulière : les automates multi-forces. Pour les caractériser nous mis en place plusieurs heuristiques et, au vu des tests, l'heuristique structurelle semble être le bon compromis entre efficacité et justesse de détection. Une autre détection peut cependant être envisagée : la détection *juste*. En effet, une composante fortement connexe peut avoir un chemin acceptant et un chemin non acceptant pour reconnaître le même mot. Dans ce cas, l'heuristique intrinsèque (qui se focalise sur les cycles élémentaires), considérera la composante comme étant forte. Du point de vue du test de vacuité, le chemin non-acceptant est inutile. Une fois tous les chemins inutiles supprimés, la force de la composante peut être recalculée. Supprimer tous les chemins inutiles est équivalent à savoir si tous les mots reconnus par un composantes sont acceptés. Ainsi, un composante peut être considérée comme faible si tous les mots reconnus sont acceptés. Étant donné que l'heuristique intrinsèque a déjà une complexité exponentielle, une telle détection n'est pas envisageable.

Toutes ces heuristiques ont permis une caractérisation des automates en fonction de leur composantes fortement connexes, et cette force a été utilisée pour optimiser les tests de vacuité basés sur un NDFS. Afin d'être indépendant du test utilisé, nous avons proposé dans ce chapitre une modification du schéma de vérification pour tirer parti des automates multi-forces. Pour cela trois automates sont construits, et une procédure de vérification spécialisée est utilisé pour tester la vacuité du produit de chaque automate avec la structure de Kripke. Comme l'union des langages des trois automates constitue le langage original, cette décomposition permet de s'assurer de la validité de cette approche.

Chapitre 8

Étude des tests de vacuité parallèles existants

Controlling complexity is the essence of computer programming.

Brian Kernighan

Sommaire

L'étape clef de l'approche par automates pour le model checking *repose sur le test de vacuité du produit synchronisé entre une structure de Kripke et l'automate de la propriété. Dans le pire cas, une exploration complète est requise ce qui est coûteux en temps et en mémoire. Pour remédier à cela, deux techniques principales existent mais ont chacune un défaut : le* Bit State Hashing *transforme la procédure de vérification en une procédure de semi-décision, tandis que le* State Space Caching *peut conduire à une explosion des temps de calcul.*

Une autre façon de combattre ces problèmes est l'accroissement du nombre d'unités de calcul. Cette augmentation peut être faite soit de manière distribuée *(plusieurs ordinateurs distants avec chacun leur mémoire) soit de manière* parallèle *(un ordinateur avec plusieurs processeurs partageant une mémoire commune). Les approches distribuées sont facilement extensibles puisqu'il suffit de rajouter un ordinateur pour augmenter la mémoire et la puissance de calcul, mais payent le coût des communications. Les approches parallèles quant à elles peuvent communiquer rapidement grâce à la mémoire partagée mais sont difficilement extensibles. L'augmentation du nombre de cœurs et des capacités mémoires des font actuellement pencher la balance du côté des algorithmes parallèles.*

La parallélisation[1] *des tests de vacuité présentée chapitre 7 est naturelle mais reste limitée à trois processeurs. Ce chapitre brosse un aperçu des tests de vacuité parallèles et distribués existants ainsi que leurs limitations ; le chapitre 9 montre comment les contourner.*

8.1 Généalogie

La figure 8.1 présente la généalogie des tests de vacuité parallèles : les boites montrent les principaux travaux tandis que les flèches indiquent leurs liens de parenté. Pour chaque algorithme nous indiquons par un simple contour s'il est basé sur un parcours DFS, et par un double contour sinon. Chaque boite est aussi annotée pour montrer si l'idée à été présentée dans un cadre parallèle ou distribué, et si elle supporte des automates forts.

Tous les tests de vacuité séquentiels présentés dans la partie II sont basés sur un parcours DFS qui permet la détection des transitions fermantes. Celles-ci sont liées à l'ordre de parcours et Reif [71] a montré en 1985 que maintenir un tel ordre dans un cadre parallèle est un problème P-complet (détails section 8.2.1). Le premier algorithme d'accessibilité distribué [80] fut naturellement basé sur un BFS qui est facilement parallélisable [70]. De nombreuses heuristiques ont ensuite été proposées pour minimiser le nombre de messages envoyés et appliquer un meilleur partitionnement de l'espace d'état entre les différents processeurs (ou unités de calcul) [57, 58].

Ce partitionnement [57, 58, 80] repose sur une fonction de hachage qui associe un état à une unité de calcul. Lors de la réception d'un état, celle-ci en calcule les successeurs avant de les transférer au processeur devant les traiter. Ce-dernier maintient la liste des états qu'il a traité pour éviter tout calcul redondant. Plusieurs heuristiques pour la fonction de hachage existent : Stern et Dill [80] proposent une fonction de partitionnement statique, Lerda et Sisto [57] suggèrent de favoriser les calculs locaux en transférant la valeur de hachage d'un état vers ses successeurs, et enfin Lerda et Visser [58] conseillent l'utilisation d'un partitionnement dynamique.

Toutes ces heuristiques sont basées sur un traitement unique de chaque état par une unité de calcul, mais nécessitent un stockage en mémoire. Certains travaux [51, 67, 79] proposent de ne plus maintenir cette information quitte à traiter plusieurs fois les mêmes états. Pour s'assurer de la terminaison ces algorithmes doivent fixer une profondeur d'exploration maximale vu qu'ils ne savent pas quels états ont déjà été traités. Cette technique, aussi appelée *Random Walk*, tire aléatoirement les successeurs d'un état afin de maximiser l'espace d'état visité. L'algorithme d'accessibilité présenté par Sivaraj et Gopalakrishnan [79] utilise un BFS séquentiel borné pour calculer les graines des BFS distribués. Krcál [51] introduit ensuite plusieurs stratégies pour construire des tests de vacuité (pour les automates forts) basés sur les *Random Walks*. Ces derniers restent néanmoins des procédures de semi-décision. Enfin Pelánek et al. [67] concluent que les résultats sur les *Random Walk* sont mitigés car ils ne permettent une exploration totale de l'automate que dans de très rares cas.

Des tests de vacuité, qui ne sont pas des procédures de semi-décision, existent néanmoins [3, 4, 11]. Suivant une idée de Lerda et Sisto [57], Barnat et al. [3] proposent de construire un NDFS distribué. Pour cela un premier DFS est lancé pour récupérer les états acceptants à partir desquels seront lancés les parcours imbriqués. À chaque fois qu'un état acceptant est découvert, il est stocké sur un nœud maître qui assure le bon ordre d'appel pour les parcours imbriqués.

1. Par la suite, nous employons le terme *paralléliser un algorithme* dans le sens *l'adapter pour qu'il utilise plusieurs processeurs.* Lorsque le besoin s'en fera sentir nous parlerons d'algorithmes *distribués* ou *parallèles.*

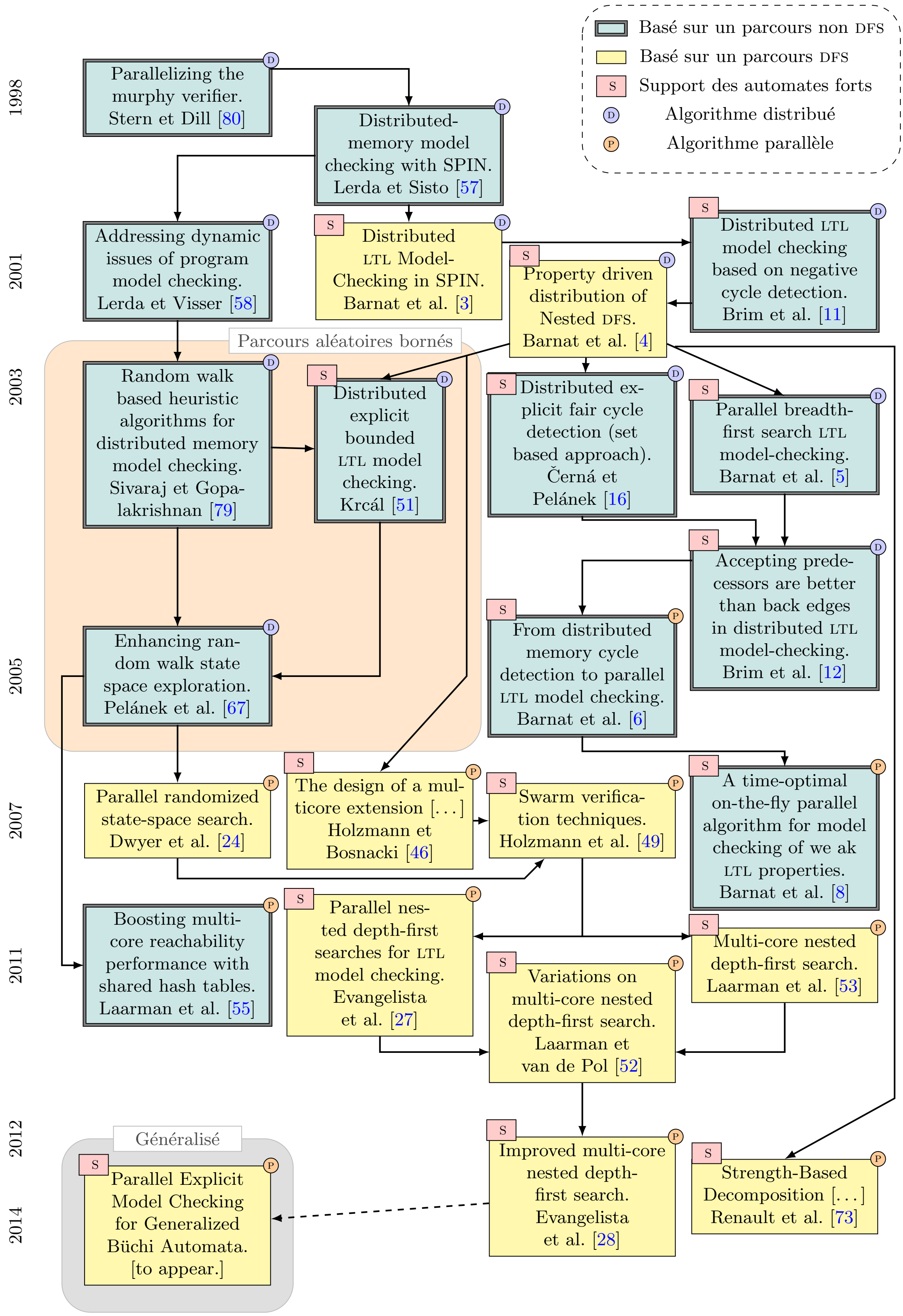


FIGURE 8.1 – Généalogie des principaux tests de vacuité parallèles et distribués.

L'algorithme présenté par Barnat et al. [4] constitue une continuité logique de cet algorithme [3]. L'idée est de lancer un unique NDFS qui va s'exécuter sur toutes les unités de calcul disponibles. Les états sont alors partitionnés en fonction de la composante fortement connexe de l'automate de la propriété à laquelle ils appartiennent, et à un instant donné seul un processeur est actif.

Bien que cet algorithme [4] soit plus efficace que celui de Barnat et al. [3], il possède de nombreux inconvénients : la fonction de partitionnement dépend de l'automate de la propriété et les différents processeurs sont sous exploités. L'absence de bonnes heuristiques pour paralléliser des tests de vacuité basés sur un NDFS a conduit à la mise en place de tests basés sur des parcours autres que DFS :

- `negc` [11] : transforme le problème du *model checking* LTL en un problème de détection de cycles négatifs. Pour cela toutes les transitions sortantes d'un état acceptant (l'algorithme est présenté sur les BA) sont étiquetée par -1 et toutes les autres par 0. L'algorithme maintient ainsi pour chaque état le plus court chemin depuis l'état initial. Si un état appartient à un cycle acceptant, il n'existe pas de tel chemin et un contre-exemple est alors détecté ;
- `bledge` [5, 6] : numérote les états par leur profondeur BFS. Dès qu'une transition remonte d'un ou plusieurs niveaux, il s'agit potentiellement d'une transition fermante : un DFS séquentiel borné est lancé pour chercher un cycle acceptant ;
- `map` [12] : suppose un ordre total entre les états afin de propager le plus petit prédécesseur d'un état. Si le plus petit prédécesseur d'un état n'est autre que lui-même il suffit de regarder s'il appartient à une composante fortement connexe acceptante. Cette appartenance est alors faite simplement en ne maintenant que le « plus petit prédécesseur acceptant ».
- `owcty` [16] : propose d'utiliser plusieurs BFS distribués pour réaliser un algorithme de point fixe. L'idée est de réduire progressivement la taille de l'espace d'état du produit pour ne garder que les états pouvant potentiellement faire partie d'un cycle acceptant. À la fin du point fixe, l'ensemble des composantes fortement connexes acceptantes est calculé : si cet ensemble est vide cela signifie qu'il n'existe pas de contre-exemple.

Les avancées matérielles ont ensuite conduit à basculer des approches distribuées aux approches parallèles pour réduire les coûts des communications. Holzmann et Bosnacki [46] proposent une adaptation de l'algorithme de Barnat et al. [3] pour les architectures dual-cores. Un premier thread cherche les états acceptants au moyen d'un DFS puis les met dans une file, un deuxième thread lance ensuite un second DFS (le parcours imbriqué) pour chaque état de cette file : c'est le premier test de vacuité parallèle basé sur un DFS partageant de l'information entre les threads.

L'intérêt de cette technique est amplifiée par les travaux de Dwyer et al. [24] qui proposent d'adapter la technique des *Random Walks* à des parcours DFS. Pour cela, chaque thread effectue une vérification complète avec un ordre de parcours qui lui est propre. Cette idée a ensuite été reprise par Holzmann et al. [49] pour la création d'un outil : SWARM. Cet outil a donné le nom de SWARMING à la technique consistant à lancer plusieurs threads avec des ordres de parcours différents.

La combinaison des deux travaux précédents [24, 46] a permis la construction de tests de vacuité parallèles avec du partage [27, 28, 52, 53]. Dans ces algorithmes, chaque thread possède son propre ordre de parcours et partage de l'information pour que les autres threads restreignent leur DFS :

- `lndfs` [53] : Laarman et al. proposent de partager les états morts. Pour cela, chaque thread maintient localement trois couleurs par états : cyan, bleu et rose. Les deux premières couleurs agissent de la même façon que pour le DFS séquentiel (cf. chapitre 3) tandis que la couleur rose permet de savoir si un état est sur la pile d'exécution du parcours imbriqué. Lorsqu'un thread détecte qu'un état est mort il peut être marqué globalement comme étant rouge : il sera ignoré par tous autres les threads ;
- `endfs` [53] : Evangelista et al. proposent une approche optimiste dans laquelle les couleurs bleues et rouges sont partagées globalement tandis les couleurs cyan et rose sont locales. Partager la couleur bleu peut conduire à des interférences entre les différents parcours. Pour éviter cela, une procédure de réparation séquentielle est lancée sur les états acceptants qui ne sont pas déjà rouges lors d'un parcours imbriqué. Ces états sont marqués comme dangereux et ne pourront passer rouges qu'après la procédure de réparation.
- `nmc-ndfs` [52] : Pour palier le coût de cette procédure séquentielle Laarman et van de Pol suggèrent de lancer un `endfs` mais d'utiliser un `lndfs` comme procédure de réparation.
- `cndfs` [28] : Bien que l'algorithme précédent combine les forces des deux tests de vacuité, il a un coût mémoire qui est deux fois plus important que chaque algorithme pris indépendemment. Evangelista et al. proposent une combinaison des deux algorithmes plus fine qui partage à la fois les couleurs bleu et rouge (détails section 3.6).

Tous ces tests de vacuité prennent en compte les automates forts mais ne supportent pas des conditions d'acceptation généralisées ce qui les pénalise en présence d'hypothèses d'équité. Pour les automates faibles et terminaux des algorithmes efficaces existent de la même manière qu'en séquentiel [8, 55].

Dans cette partie, nous nous intéressons à l'élaboration d'un algorithme supportant les automates de Büchi généralisés, i.e lorsque l'automate de la propriété est fort et qu'il y a des hypothèses d'équité.

Note : Nous ne nous intéressons ici qu'à la vérification de propriétés de logiques temporelles linéaires. D'autres approches existent néanmoins pour les logiques arborescentes [75] et les classifications de la section suivantes y sont applicables.

8.2 Classification des algorithmes

Cette section s'intéresse à la mise en place d'une classification des tests de vacuité parallèles en prenant en compte tous les paramètres pouvant les impacter.

8.2.1 Problème de l'ordre postfixe

Les tests de vacuité parallèles ont été massivement étudiés ces dernières années et les meilleures performances sont actuellement obtenues en couplant parcours NDFS, SWARMING et partage d'information entre les threads [28]. L'efficacité de cette combinaison vient essentiellement du parcours DFS qui permet la détection des transitions fermantes. Ces tests doivent néanmoins gérer le problème de l'ordre postfixe qui en détermine la correction : les états doivent être marqués comme rouges dans le second parcours en suivant l'ordre postfixe du parcours principal (i.e. l'ordre dans lequel les états sont dépilés de la pile DFS). Dans un cadre parallèle, deux threads peuvent avoir des ordres différents et les parcours imbriqués peuvent alors interférer.

Considérons une parallélisation naïve du NDFS présenté chapitre 3 dans laquelle la variable *coloredset* est partagée par tous les threads. Les états sont colorés globalement par chaque thread qui possède son propre ordre de parcours [2]. Dans un tel algorithme un thread peut marquer un état comme rouge alors qu'il n'a pas visité tous ses successeurs : certains contre-exemples peuvent être manqués.

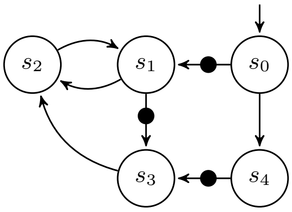
Exemple.
La figure 8.2 montre un cas dans lequel les parcours rouges de deux threads interfèrent. Pour chaque étape est représenté l'extrait utile des piles DFS des parcours bleus de chaque thread (resp. $dfs1$ et $dfs2$). Lors qu'un des threads est en train d'effectuer un parcours rouge nous présentons la pile DFS de ce parcours(resp. $reddfs1$ pour le premier thread et $reddfs2$ pour le second).

L'étape 0	:	présente le TBA (du produit synchronisé) utilisé comme exemple ;
Les étapes 1 à 4	:	décrivent l'exécution du premier thread qui va colorier en bleu les états s_0, s_1, s_2 et s_3. Notons que la détection de la transitions fermante (s_2,s_1) ne lance pas de parcours imbriqué puisqu'elle n'est pas acceptante ;
L'étape 5	:	montre la commutation des deux threads. Le deuxième thread prend alors la main et marque les états s_0 et s_4 en bleu ;
Les étapes 6 et 7	:	montrent la détection de la transition (s_4, s_3) qui conduit au déclenchement d'un parcours imbriqué puisque l'état s_3 est bleu. Ces deux étapes montrent l'évolution de ce parcours et la coloration en rouge des états s_3 puis s_2 ;
Les étape 8 et 9	:	montrent la reprise d'exécution du premier thread. Lors du backtrack, il détecte que la transition (s_1, s_3) est acceptante et lance un second parcours. Comme tous les successeurs de s_3 sont rouges ce parcours s'arrête immédiatement. À l'étape 9, la détection de la transition acceptante (s_0, s_1) conduit alors à déclencher un autre parcours pour trouver un cycle autour de l'état s_1. L'état s_1 est alors marqué rouge et le parcours imbriqué s'arrête.

À la fin de l'étape 9, le parcours imbriqué du deuxième thread n'est pas terminé mais aucun contre-exemple ne peut être détecté car l'état s_1 est déjà marqué rouge et sera donc ignoré par tous les autres parcours.

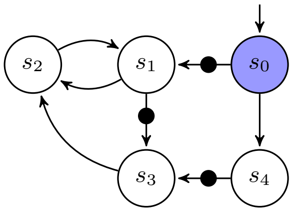
Dans cet exemple, un partage global de la couleur cyan peut conduire à la détection d'un

2. Ici nous ne considérons pas l'optimisation qui consiste à utiliser la couleur cyan : une simple coloration bleue et rouge suffit pour montrer le problème de l'ordre postfixe. De même nous n'utilisons pas ici l'optimisation permettant de savoir si tous les successeurs d'un état sont rouges.



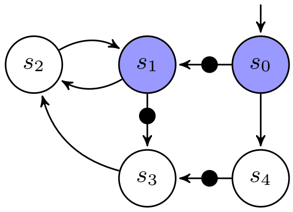
dfs1 :					
dfs2 :					

Étape 0.
L'automate à vérifier



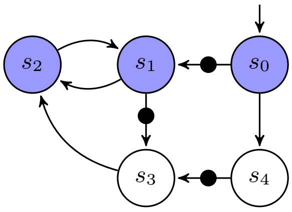
dfs1 :	s_0				
dfs2 :					

Étape 1.
Thead 1 marque s_0 bleu



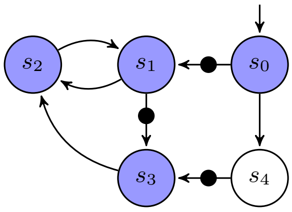
dfs1 :	s_0	s_1			
dfs2 :					

Étape 2.
Thead 1 marque s_1 bleu



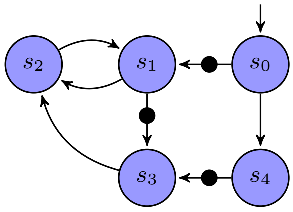
dfs1 :	s_0	s_1	s_2		
dfs2 :					

Étape 3.
Thead 1 marque s_2 bleu



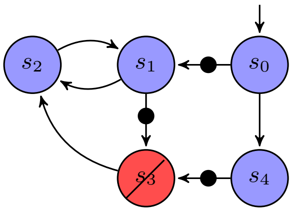
dfs1 :	s_0	s_1	s_3		
dfs2 :					

Étape 4.
Thead 1 marque s_3 bleu



dfs1 :	s_0	s_1	s_3		
dfs2 :	s_0	s_4			

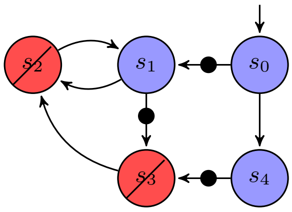
Étape 5.
Thead 2 marque s_0, puis s_4 bleu



dfs1 :	s_0	s_1	s_3		
dfs2 :	s_0	s_4			

reddfs2 :	s_3		

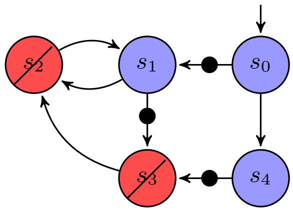
Étape 6.
Thead 2 marque s_3 rouge
lors d'un $\texttt{UPDATE}_{\text{NDFS}}$



dfs1 :	s_0	s_1	s_3		
dfs2 :	s_0	s_4			

reddfs2 :	s_3	s_2	

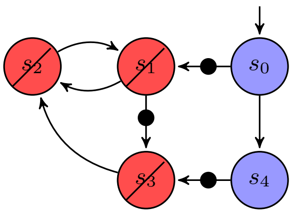
Étape 7.
Thead 2 marque s_2 rouge
dans le parcours imbriqué



dfs1 :	s_0	s_1			
dfs2 :	s_0	s_4			

reddfs3 :	s_3	s_2	

Étape 8.
Thead 1 dépile s_3
comme ses successeurs sont déjà rouges, rien ne se passe



dfs1 :	s_0				
dfs2 :	s_0	s_4			

reddfs2 :	s_3	s_2	

Étape 9.
Thead 1 dépile s_1
lors d'un $\texttt{POP}_{\text{NDFS}}$

Comme la transition (s_0, s_1) est acceptante, un parcours imbriqué est lancé. Cependant tous les successeurs sont rouges et il n'y aura pas de détection de cycle acceptant.

Figure 8.2 – Problème de l'ordre postfixe pour une parallélisation naïve du NDFS.

contre-exemple erroné. À l'étape 5, les états s_1 et s_4 sont de cette couleur et l'existence d'une transition entre ces deux états provoquerait une détection erronée. Partager la couleur cyan conduit à la détection de faux contre-exemples alors que le problème de l'ordre postfixe conduit seulement à manquer des contre-exemples.

8.2.2 Classification des algorithmes

Plusieurs solutions ont été proposées pour éviter le problème de l'ordre postfixe et partager le maximum d'informations entre les différents threads. Laarman et al. [53] proposent que cyan soit une couleur locale et que le parcours imbriqué soit bloqué tant que d'autre parcours imbriqués visitent les mêmes états. Evangelista et al. [27] proposent de détecter les états problématiques pour les traiter au sein d'une procédure séquentielle. Enfin Evangelista et al. [28] suggèrent de bloquer les parcours imbriqués en attendant que tous les états problématiques aient été colorés en rouge [3].

Face à ces multiples stratégies et aux différents types de parcours utilisés il est difficile de classifier les tests de vacuité parallèles. L'unique classification existante a été introduite par Barnat et al. [8] pour caractériser le degré de compatabilité d'un algorithme avec la construction de l'automate à la volée. Cette mesure permet une comparaison efficace des algorithmes basés sur un BFS et repose sur la notion de *détection prématurée*, i.e. la détection d'un cycle acceptant avant que l'algorithme ne détecte l'absence de nouvel état à explorer.

Définition 22 – On-the-flyness classification (Barnat et al. [8])

- $\mathbf{L}_0$ Il existe un automate contenant un cycle acceptant pour lequel l'algorithme ne terminera jamais de manière prématurée ;
- $\mathbf{L}_1$ Pour tous les automates contenant un cycle acceptant, l'algorithme peut terminer de manière prématurée mais sans garantie ;
- $\mathbf{L}_2$ Pour tous les automates contenant un cycle acceptant, l'algorithme garanti une terminaison prématurée.

La classification précédente permet de détecter si un cycle acceptant peut être découvert de manière prématurée mais ne fournit aucune information sur la redondance de travail effectuée entre les threads. La redondance est induite par le nombre de fois où un état est traité, i.e. le nombre de fois ou la fonction de transition est sollicitée pour le générer. Nous proposons donc la classification suivante :

Définition 23 – Classification de la redondance

- $\mathbf{R}_0$ Un état est traité au plus k fois ($k \in \mathbb{N}$, $k > 0$), où k dépend de l'automate en entrée ;
- $\mathbf{R}_1$ Un état est traité au plus k fois ($k \in \mathbb{N}$, $k > 1$), où k est fixé indépendemment de l'automate en entrée ;
- $\mathbf{R}_2$ Un état est traité au maximum une fois pour n'importe quel automate en entrée.

3. Nous verrons section 8.3.2 comment cet algorithme traite l'exemple montré section 8.2.1.

Le nombre de fois qu'un état est traité par l'algorithme est une bonne indication du passage à l'échelle de l'algorithme lors de l'augmentation du nombre de threads. Dans la classification précédente le degré $\mathbf{R}_2$ constitue l'optimal puisqu'un simple parcours de l'automate permet d'en détecter la vacuité. Pour avoir une idée de l'accélération possible d'un algorithme, cette indication doit être complétée par :

1. la présence de synchronisations ou de procédures de réparations ;
2. le nombre maximal de cœurs supportés [4] ;
3. le support de conditions d'acceptations généralisées.

La table 8.1 permet une comparaison des tests de vacuité en fonction des critères mentionnés ci-dessus. Notons tout d'abord que ces caractérisations sont données dans le pire cas, i.e. pour des automates forts. Pour des automates faibles ou terminaux, des variations peuvent exister : par exemple, l'algorithme `owcty-map` a une complexité $\mathbf{L}_2$ pour les automates faibles.

	Algorithme	Ref	Comp. à la volée	Redon--dance	Pas de réparations	Pas de synchronisations	Max cœurs	Géné--ralisé
BFS	`negc`	[11]	L_0	R_0	✓	✓	N	
	`bledge`	[5]	L_0	R_0	✓		N	
	`owcty`	[16]	L_0	R_0	✓	✓	N	
	`map`	[12]	L_1	R_0	✓	✓	N	
	`bledge-otf`	[6]	L_2	R_0	✓		N	
	`owcty-map`	[8]	L_1	R_0	✓	✓	N	
NDFS	`2-ndfs`	[46]	L_2	R_1	✓	✓	2	
	`endfs`	[27]	L_2	R_1		✓	N	
	`lndfs`	[53]	L_2	R_1	✓		N	
	`nmc-ndfs`	[52]	L_2	R_1			N	
	`cndfs`	[28]	L_2	R_1	✓		N	
DFS	`Dijkstra`	Chapitre 9	L_2	R_1	✓	✓	N	✓
	`Tarjan`	Chapitre 9	L_2	R_1	✓	✓	N	✓
	`Mixed`	Chapitre 9	L_2	R_1	✓	✓	N	✓

Table 8.1 – Comparaison des tests de vacuité parallèles. Notons que certains algorithmes ont été présentés dans un contexte distribué ou n'utilisant pas des structures lock-free (typiquement les files). Pour être juste et en accord avec les progrès matériels récents, nous classons les algorithmes en supposant l'utilisation de telles structures en mémoire partagée (comme suggéré par Barnat et al. [7]).

Il apparaît que tous les tests de vacuité basés sur un parcours BFS ont une redondance $\mathbf{R}_0$: cela est du à l'utilisation d'algorithmes de points fixes dont le nombre d'itération dépend de l'automate. Tous les algorithmes basés sur un NDFS peuvent, quant à eux, détecter les transitions fermantes et ne requièrent donc pas de point fixe : ils ont des complexités $\mathbf{R}_1$. Les seuls algorithmes existants qui ont une complexité $\mathbf{R}_2$ sont des algorithmes d'accessibilité et ne supportent donc pas tous les types d'automates : ils ne sont donc pas mentionnés ici. De plus, notons que l'existence de structures *lock-free* (i.e. qui ne requièrent pas de synchronisation entre les threads) et de variables atomiques (dont les modifications peuvent être concurrentes) permet de construire des algorithmes parallèles qui n'ont pas recours à des mécanismes de synchronisation.

4. Cette condition est nécessaire mais pas suffisante à assurer un bon passage à l'échelle.

Tous les algorithmes basés sur un NDFS, à l'exception de `2-ndfs` [46], requièrent soit des points de synchronisation, soit des procédure de réparation. L'algorithme `2-ndfs` n'en a pas besoin mais il ne supporte pas plus de deux threads. L'étude de ce tableau montre que :

1. il n'existe pas d'algorithme parallèle généralisé ;
2. qu'aucun algorithme basé sur un NDFS ne peut se passer de procédures de réparation ou de procédure de synchronisation pour maintenir l'ordre postfixe évoqué dans la section précédente.

Ces deux critères sont pourtant essentiels à la mise en place d'un algorithme parallèle passant facilement à l'échelle et qui supporte l'équité [5]. Néanmoins il a été montré par Evangelista et al. [28] que les meilleures performances sont obtenues par l'algorithme `cndfs` qui permet de limiter au maximum les points de synchronisation.

8.3 Tests de vacuité parallèles basés sur un NDFS

Cette section détaille le test de vacuité existant réputé le plus efficace à savoir l'algorithme `cndfs` d'Evangelista et al. [28] qui est le résultat de la combinaison de trois algorithmes (`endfs`, `lndfs`, et `nmc-ndfs`).

8.3.1 Modification de l'algorithme principal

De la même manière que pour les algorithmes séquentiels, les tests de vacuité parallèles peuvent être lancés au travers d'une unique procédure principale présentée par l'algorithme 4. Comme tous les tests parallèles sont basés sur un parcours DFS [6] couplé avec du SWARMING, la procédure `parallel_main` impose pour tous les threads la politique de parcours RANDOM. La principale différence par rapport à la procédure principale pour les tests séquentiels (algorithme 1, page 52), concerne les lignes 2 et 3 : tous les threads sont mis en parallèle chacun ayant sa propre politique de parcours.

```
1 parallel_main(str : Strategy)
2     EC(str, RANDOM) || ... || EC(str, RANDOM)
3     Wait for all threads to finish
```

Algorithme 4: Procédure principale pour les tests de vacuité parallèles.

8.3.2 Détail de l'algorithme `cndfs`

Cet algorithme combine les avantages de `lndfs` et `endfs` mais nécessite des points de synchronisation pour s'assurer d'une coloration dans l'ordre postfixe. La stratégie 10 présente le raffinement de la stratégie 4 nécessaire à la mise en place de cet algorithme : il s'agit de la première fois que cet algorithme est présenté sur les TBA [7]. Nous détaillerons les modifications qui ont été faites pour permettre ce support des TBA.

5. Le chapitre 9 montre comment la mise en place d'un tel algorithme est possible au moyen de structures *lock-free*.
6. On ne s'intéresse pas ici à ceux basés sur un parcours BFS qui offrent de moins bonnes performances.
7. Une preuve complète de cet algorithme a été présentée par Laarman [54].

```
Structures supplémentaires :
  struct color { isBlue : bool, isRed : bool }
  struct Step {src : Q,     succ : 2^Δ,
               acc : 2^F,  allred : bool} // Refinement of Step of Algo. 1

Variables Partagées supplémentaires :
  coloredset : map of Q ↦ ⟨c : color⟩

Variables Locales supplémentaires :
  cyanset : hashset of ⟨ s ∈ Q ⟩
  R_acc : hashset of ⟨ s ∈ Q ⟩
  R_st : hashset of ⟨ s ∈ Q ⟩

PUSH_cndfs (acc ∈ 2^F, q ∈ Q) → int
  cyanset.insert(⟨ q ⟩)
  coloredset.insert(⟨ q, ⟨⊥, ⊥⟩ ⟩)
  dfs.push(⟨ q, succ(q), acc, ⊤ ⟩)

GET_STATUS_cndfs (q ∈ Q) → Status
  if ¬ cyanset.contains(q) ∧
    (¬ coloredset.contains(q) ∨ ¬ coloredset.get(q).c.isBlue) then
      return UNKNOWN
  if coloredset.get(q).c.isRed then
      return DEAD
  return LIVE

UPDATE_cndfs (acc ∈ 2^F, dst ∈ Q)
  if acc= 2^F then
      if cyanset.contains(dst) then
          report Accepting cycle detected!
      else
          mark_red(dst)            // Details page 126
  else
      dfs.top().allred ← ⊥

POP_cndfs (s ∈ Step)
  dfs.pop()
  coloredset.get(s.src).c.isBlue ← ⊤
  cyanset.remove(s.src)
  if s.allred= ⊤ then
      coloredset.get(s.src).c.isRed ← ⊤
  else if s.acc = 2^F then
      mark_red(s.src)            // Details page 126
  else if ¬ dfs.empty() then
      dfs.top().allred ← ⊥
```

Stratégie 10: Adaptation aux TBA du `cndfs` présenté par Evangelista et al. [28]

```
mark_red(s : Q)
    R_st.clear()
    R_acc.clear()
    R_acc.insert(s)
    nested_dfs(s)
    await ∀ s' ∈ R_acc : s' ≠ s  ⟹  coloredset.get(s').c.isRed
    forall the s' in R_st ∪ { s } do
        coloredset.get(s').c.isRed ← ⊤

// Also known as Red-DFS
nested_dfs(q ∈ Q)
    forall the Transition t ∈ succ(q) do
        if cyanset.contains(t.dst) then
            report Accepting cycle detected!
        if t.acc = 𝓕 then
            if t.src ∈ R_st then
                R_st ← R_st \ t.src
            R_acc ← R_acc ∪ t.src
        if (t.dst ∉ R_st ∪ R_acc) ∧ ¬ coloredset.get(t.dst).c.isRed then
            nested_dfs(t.dst)
```

Algorithme 5: Cœur de la parallélisation de l'algorithme `cndfs`

Dans cet algorithme les couleurs rouges et bleues d'un état sont partagées par tous les threads. Contrairement au NDFS séquentiel présenté section 3 un état peut être simultanément bleu et rouge mais seuls les états bleus peuvent devenir rouges. La structure *color* (ligne 2) permet de stocker cette information via seulement deux bits. La table de hachage lock-free *coloredset* permet le partage des couleurs rouges et bleues entre les threads. La couleur cyan quant à elle est intimement liée à l'ordre de parcours et doit être stockée localement. Pour plus de clarté, la stratégie 10 utilise un ensemble dédié *cyanset* pour stocker cette information ; dans la pratique la structure *color* peut être modifiée pour réserver un bit par thread.

Lors d'un $\texttt{PUSH}_{cndfs}$ les états sont d'abord insérés localement dans *cyanset* avant d'être insérés globalement dans *coloredset*. Lors de cette insertion globale, si l'état est déjà présent dans la table de hachage partagée, alors aucune insertion n'est effectuée et les valeurs *isRed* et *isBlue* associées restent inchangées. L'état est ensuite inséré dans la pile *dfs* du thread considéré exactement comme l'algorithme de la section 3.

La méthode $\texttt{GET\_STATUS}_{cndfs}$ permet ensuite de connaître le statut d'un état. Tous les états cyans sont nécessairement vivants puisqu'ils sont sur la pile d'exécution du parcours principal du thread. Pour une détection des transitons fermantes les états bleus doivent eux aussi être considérés comme vivants. Les états rouges, en revanche, ne peuvent faire partie d'aucun contre-exemple et sont morts pour l'ensemble des threads. Tous les autres états (non cyans, non bleus) sont inconnus et restent à explorer.

Les méthodes $\texttt{UPDATE}_{cndfs}$ et $\texttt{POP}_{cndfs}$ ne varient par rapport à la stratégie 4 (page 57) que par les lignes 26, 31, 32 35 et 37. Les variations aux lignes 31 et 37 prennent en compte qu'un état est caractérisé maintenant à la fois par sa couleur bleue et par sa couleur rouge. Lors d'un $\texttt{POP}_{cndfs}$ tous les états doivent être marqués bleus, car les états ne devenir rouges qu'une fois après avoir été marqués bleus. La modification de la ligne 32 sert, quant à elle, à ne plus considérer un état comme cyan à partir du moment où il a été globalement marqué bleu et ne se situe plus sur la pile *dfs* du parcours principal. Les modifications des lignes 26 et 36 constituent le cœur de la parallélisation et sont regroupées sous l'appel à la fonction `mark_red`.

L'algorithme 5 présente cette méthode qui permet à la fois de détecter les contre-exemples, mais aussi de colorer les états en rouges en respectant l'ordre postfixe. Un appel à la méthode `nested_dfs` (ligne 5) collecte tous les états non rouges accessibles depuis l'état ayant déclenché la procédure `mark_red`. Cette procédure détecte un contre-exemple dès qu'un état cyan est rencontré lors de l'exploration. La collection des états se fait au travers des variables R_{acc} et R_{st} contenant respectivement tous les états sources d'une transition acceptante (ainsi que l'état ayant déclenché la procédure `mark_red`) et tous ceux qui ne le sont pas. Si un état est à la fois la source d'une transition acceptante et d'une transition non-acceptante il sera donc stocké dans R_{acc}. Ces deux ensembles servent aussi à orienter le parcours imbriqué pour qu'il ne visite que des états non-rouges n'étant pas déjà dans $R_{st} \cup R_{acc}$.

À la fin du parcours imbriqué, la procédure `mark_red` attend que tous les états de R_{acc} (à l'exception de celui ayant déclenché `mark_red`) soient marqués comme rouges (ligne 6). Cela permet de s'assurer que l'ordre postfixe est respecté afin de palier le problème évoqué en section précédente. Intuitivement, un état source d'une transition acceptante ne peut être considéré mort que par le thread ayant lancé un parcours imbriqué depuis cette transition. Une fois que l'ordre postfixe est assuré, un thread peut marquer tous les états collectés comme rouges (ligne 7 et 8).

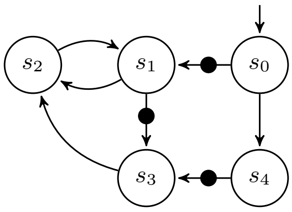
Dans le pire cas, l'ensemble $R_{st} \cup R_{acc}$ peut contenir l'ensemble des états de l'automate. Notons que l'algorithme original basé sur les BA : il n'utilise qu'un seul ensemble R pour collecter les états lors du parcours imbriqué. Cela est possible car les marques d'acceptation sont portées par les états et non par les transitions.

Note : Les optimisations présentées au chapitre 7 pour limiter la profondeur du parcours imbriqué sont applicables ici.

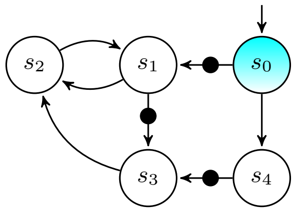
8.3.3 Déroulement de l'algorithme

Exemple.
La figure 8.3 reprend l'exécution de la figure 8.2 et montre comment l'algorithme `cndfs` permet de palier le problème alors mis en évidence. L'étape 1 montre l'automate à vérifier. Les étapes 2 à 4 présentent l'exploration par le premier thread des états s_0, s_1, s_2 : ces états sont marqués localement cyan : ils sont donc représentés sur la figure en dégradé. À la fin de l'exploration de l'état s_2, comme tous ses successeurs ne sont pas rouges, il est marqué globalement bleu (étape 5).

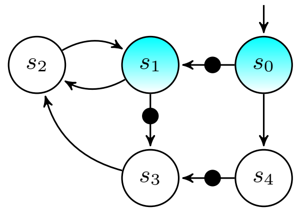
Le deuxième thread prend ensuite la main et marque localement les états s_0 et s_4 comme cyan aux étapes 6 et 7. Aux étapes 8 et 9, le premier thread reprend son exécution pour marquer l'état s_3 tout d'abord comme cyan localement puis comme bleu globalement. Ce marquage global est fait lorsque l'état s_3 est dépilé de la pile *dfs*, comme la transition (s_1, s_3) est acceptante, une procédure `mark_red` est ensuite lancée sur l'état s_3. Avant que cette



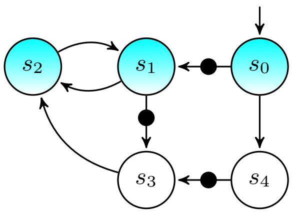
Étape 1. L'automate à vérifier



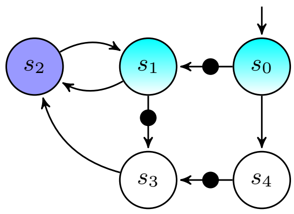
Étape 2. Thread 1 marque loca- -lement s_0 comme cyan



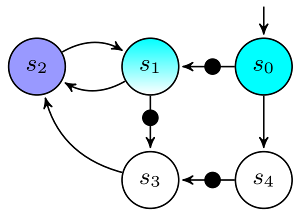
Étape 3. Thread 1 marque loca- -lement s_1 comme cyan



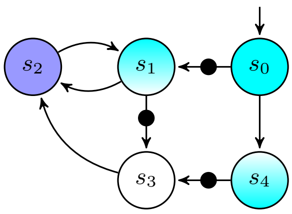
Étape 4. Thread 1 marque loca- -lement s_2 comme cyan



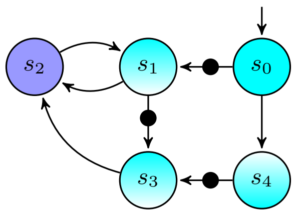
Étape 5. Thread 1 marque globa- -lement s_2 comme bleu



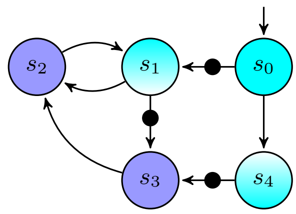
Étape 6. Thread 2 marque loca- -lement s_0 comme cyan



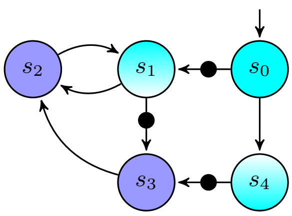
Étape 7. Thread 2 marque loca- lement s_4 comme cyan



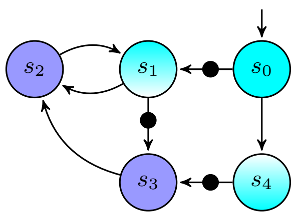
Étape 8. Thread 1 marque loca- lement s_3 comme cyan



Étape 9. Thread 1 marque globa- lement s_3 comme bleu



Étape 10. Thread 2 collecte :
R_{acc}=$\{s_3, s_1\}$, R_{st}=$\{s_2\}$
Attente s_1 passe rouge



Étape 11. Thread 1 collecte :
R_{acc}=$\{s_3, s_1\}$, R_{st}=$\{s_2\}$

Dès que s1 est rencontré (par le Thread 1) comme il est cyan, un contre-exemple est détecté !

Figure 8.3 – Déroulement d'un exemple sur l'algorithme `cndfs`

procédure ne s'exécute, le deuxième thread reprend la main et détecte que l'état s_3 est bleu, il lance alors lui aussi une procédure `mark_red` sur cet état.

Le deuxième thread lance donc la procédure `nested_dfs` pour collecter tous les états non rouges accessibles depuis l'état s_3. A la fin de cette étape R_{acc} contient les états s_3 et s_1. Le deuxième thread ne peut alors pas marquer ces états comme rouge avant que l'état s_3 soit lui même marqué comme rouge. Ainsi l'ordre postfixe est respecté. C'est donc le premier thread qui doit marquer l'état s_1 comme rouge. Cet état ne sera jamais marqué rouge car la procédure `nested_dfs` du premier thread découvre que l'état s_1 est marqué cyan ce qui implique l'existence d'un cycle acceptant autour de l'état s_3.

Par opposition à l'exemple de la figure 8.2, l'instruction **await** de la stratégie 10 force l'ordre de coloration des états en rouge : les états sources d'une transition acceptante ne peuvent être marqué comme rouge que par les threads les ayant vu dans le parcours principal.

8.4 Conclusion

Dans ce chapitre, nous avons vu que de nombreux tests de vacuité parallèles existent. Ces derniers sont difficilement comparables car ils n'utilisent pas forcément les même types de parcours ou les mêmes mécanismes pour s'assurer de la justesse des informations partagées. Pour remédier à cela, nous avons proposé une classification et mis en avant plusieurs critères qui semblent nécessaires à une comparaison objective de ces algorithmes.

L'algorithme le plus efficace connu est l'algorithme `cndfs` qui combine parcours NDFS, SWARMING et partage des couleurs associées aux états entre les threads. L'utilisation d'un parcours DFS lui permet une détection des transitions fermantes et d'avoir une redondance $\mathbf{R}_1$. Ce test de vacuité possède néanmoins trois inconvénients majeurs : (1) il ne supporte pas les ensembles d'acceptation multiples, (2) il nécessite l'utilisation d'une procédure de synchronisation qui est un frein à une parallélisation efficace, et (3) il ne restreint pas le parcours imbriquée aux seules composantes acceptantes (comme proposé par Edelkamp et al. [26]).

Actuellement, aucun test de vacuité ne s'est intéressé à la mise en place d'un algorithme basé sur un DFS qui supporte les ensembles d'acceptation multiples et qui ne requiert pas de procédure de synchronisation. Dans le chapitre suivant, nous montrons que la construction d'un tel algorithme est possible.

Chapitre 9

Tests de vacuité généralisés parallèles

An algorithm must be seen to be believed.

Donald Knuth

Sommaire

Dans le chapitre précédent, nous avons vu que les tests de vacuité parallèles les plus efficaces combinent SWARMING *et parcours* NDFS*. Chaque thread effectue une exploration qui lui est propre et partage l'information qu'il collecte. Comme cette information est liée à l'ordre de parcours, sa validité repose sur des procédures de réparation ou de synchronisation. Celles-ci peuvent avoir un impact négatif sur le passage à l'échelle, et le seul test de vacuité parallèle pouvant s'en passer ne peut utiliser plus de deux cœurs [46].*

Tous les tests de vacuité parallèles existants ne supportent que des automates non-généralisés, et la vérification de propriétés sous hypothèses d'équité ne peut être faite efficacement (comme nous l'avons vu au chapitre 6). Le support d'automates généralisés nécessite habituellement un calcul des composantes fortement connexes. Les seuls algorithmes parallèles effectuant ce calcul ont des performances mitigées dues à l'utilisation d'un parcours BFS *(qui ne détecte pas les transitions fermantes) et de procédures de synchronisation.*

Dans ce chapitre nous proposons les premiers tests de vacuité parallèles généralisés ne nécessitant ni procédures de synchronisation, ni procédures de réparation. Ces tests, s'appuient sur des architectures multi-cœurs et partagent de l'information entre les threads : ils viennent concurrencer ceux présentés dans le chapitre précédent. Ils combinent ainsi plusieurs idées : l'utilisation d'un parcours DFS *pour détecter facilement les transitions fermantes, l'utilisation d'un union-find pour partager l'information et enfin du* SWARMING *pour s'assurer d'une bonne répartition de l'exploration effectuée par les threads.*

9.1 Idée générale

Dans les algorithmes `lndfs`, `endfs`, `nmc-ndfs`, et `cndfs` une partie de l'information partagée est liée à l'état d'un thread puisqu'elle est dépendante de l'ordre de parcours. Cette information peut donc varier en fonction des threads. Par exemple, dans l'algorithme `cndfs`, un état peut être marqué globalement bleu alors que les autres threads n'ont même pas la connaissance de cet état. Comme ces états conditionnent les parcours imbriqués, certains threads peuvent déclencher un second parcours (et se bloquer) sur des états qui leurs sont inconnus.

Partager une information *stable*, i.e. qui liée à la structure de l'automate plutôt qu'à l'état d'un thread, est une condition suffisante à la construction d'algorithmes parallèles sans synchronisation ni réparation. Une information est donc dite stable si elle est valide quel que soit l'ordre de parcours considéré. On distingue trois types d'informations stables :

(i) deux états sont dans la même composante fortement connexe ;

(ii) une marque d'acceptation est présente dans une composante fortement connexe ;

(iii) un état et tous ses successeurs directs ou indirects ne peuvent appartenir à une exécution acceptante : ils sont *morts*.

Dans l'algorithme `cndfs` seule l'information sur la vivacité d'un état (iii) peut être partagée car l'utilisation d'un parcours NDFS ne permet pas de récupérer des informations sur les composantes fortement connexes. Les tests de vacuité séquentiels basés sur les algorithmes de Tarjan et Dijkstra capturent naturellement ces informations et sont donc de bons candidats à la construction de tests de vacuité parallèles partageant de l'information stable. Pour cela, chaque thread effectue un test de vacuité séquentiel, et partage les informations stables dès qu'elles sont calculées. Il suffit ensuite que chaque thread consulte régulièrement les informations publiées pour adapter son parcours et accélérer la détection de cycles acceptants. Lorsqu'un état est globalement marqué comme étant mort il peut être ignoré par tous les threads : cette information ne pourra plus évoluer jusqu'à la fin du test de vacuité. De même, l'information que deux états sont dans la même composante fortement connexe ne peut évoluer mais des nouvelles marques d'acceptation ou de nouveaux états peuvent être rajoutés à cette composante. Ces états peuvent ensuite être marqués comme morts pour restreindre les autres parcours.

Dans le chapitre 5 nous avons proposé l'utilisation d'une structure d'union-find aussi bien pour stocker l'appartenance d'un état à une composante fortement connexe que pour marquer comme mort un ensemble d'états en une unique opération. Dans un cadre parallèle, l'union-find peut être partagé et cette opération peut être faite atomiquement (i.e. en une instruction machine), ce qui permet de restreindre immédiatement le parcours de tous les autres threads. Enfin, chaque élément de cette structure peut stocker un ensemble de marques d'acceptation en plus de son lien de parenté : cela permet de représenter l'ensemble des marques découvertes dans la composante fortement connexe.

Lors d'un `makeset`, l'élément inséré est associé à un ensemble d'acceptation vide. Si un thread essaye de créer un élément qui existe déjà, cette création échoue : ainsi aucune information n'est perdue. Cette opération peut être faite de manière atomique. L'opération `unite` permet d'unir deux éléments (ou classes), et cette opération peut aussi être faite de manière atomique. Ainsi dès que deux états sont détectés comme étant dans la même composante fortement connexe ils sont unis et cette information est disponible immédiatement pour tous les threads.

L'information que deux états sont dans la même composante fortement connexe est généralement couplée à un ensemble de marques d'acceptation elles aussi présentes dans la composante. L'opération `unite` est ainsi étendue pour prendre en argument supplémentaire un ensemble de marques d'acceptation. L'union de deux partitions combine alors à la fois les marques d'acceptation associées à chaque classe mais aussi les marques passées en argument de `unite`. Une fois l'union de deux classes réalisées, le représentant d'une classe possède l'union de toute les marques et cette information est disponible pour tous les threads au travers de la valeur de retour de la méthode `unite` [1].

De la même manière que pour le chapitre 5, marquer un état comme mort peut être fait de manière atomique en utilisant une partition spéciale contenant l'état artificiel *alldead* toujours associé à un ensemble d'acceptation vide.

Discussion. Nous utilisons ici une structure d'union-find qui est *lock-free* et dont la structure diffère de celle présentée au chapitre 5. Elle est composée d'une table de hachage *lock-free* identique à celle utilisée par Laarman et al. [53]. Cette table stocke des états auxquels sont associés un nœud. Ce nœud constitue alors un point d'entrée sur la liste (simplement chaînée) des liens de parentés. Comme la mise à jour de cette liste peut être faite atomiquement au moyen d'opérations *compare-and-swap*, l'ensemble de la structure est *lock-free*.

9.2 Parallélisation de l'algorithme de Tarjan

Cette section montre comment le test de vacuité séquentiel basé sur l'algorithme de Tarjan peut être transformé en un algorithme parallèle. Les modifications à apporter à l'algorithme original sont minimes puisqu'il s'agit simplement de modifications liées à la publications ou à la prise en compte d'informations stables.

9.2.1 Détails de l'algorithme

Cette parallélisation est présentée par la stratégie 11 et s'intègre dans le cadre de notre DFS générique. Chaque thread est instancié par la procédure 7 page 135 (avec la stratégie TarjanPar) et exécute son propre DFS en choisissant les successeurs de manière aléatoire [2]. Chaque thread effectue son propre test de vacuité et stocke localement l'information qui n'est pas stable. De la même manière que pour l'algorithme original, chaque état vivant est associé à un identifiant dans *livenum*, et tous les états vivants qui ne sont plus sur la pile DFS sont stockés dans la pile *live* (optimisation de Nuutila et Soisalon-Soininen [61]). La seule variable partagée par tous les threads (en dehors de la variable *stop* utilisée comme condition d'arrêt) est la structure d'union-find modifiée pour stocker les ensembles d'acceptation et pour être *thread-safe*.

1. Plus de détails sur l'implémentation d'une telle structure peuvent être trouvés en annexe A.
2. Cette procédure ne varie par rapport à l'algorithme 7 que par l'initialisation de la strucutre d'union-find à la ligne 2.

```
Structures supplémentaires :
  struct Step {src : Q,    succ : 2^Δ,
               acc : 2^F,  pos : int} // Refinement of Step of Algo 1
Variables Locales supplémentaires :
  live : stack of ⟨ Q ⟩
  livenum : map of Q ↦ ⟨p : int⟩
  llstack : pstack of ⟨ p : int⟩ // Described Section 4.4
Variables Partagées supplémentaires :
  uf : union-find of ⟨ Q ∪ alldead, 2^F ⟩

PUSH_TarjanPar(acc ∈ 2^F, q ∈ Q) → int
  uf.makeset(q)
  p ← livenum.size()
  livenum.insert(⟨ q, p ⟩)
  llstack.pushtransient(p)
  dfs.push(⟨ q, succ(q), acc, p ⟩)

GET_STATUS_TarjanPar(q ∈ Q) → Status
  if livenum.contains(q) then
    return LIVE
  else if uf.contains(q) ∧ uf.sameset(q, alldead) then
    return DEAD
  else
    return UNKNOWN

UPDATE_TarjanPar(acc ∈ 2^F, dst ∈ Q)
  p ← llstack.pop(dfs.top().pos)
  llstack.pushnontransient(min(p, livenum.get(d)))
  a ← uf.unite(dst, dfs.top().src, acc)
  if a = F then
    stop ← ⊤
    report Accepting cycle detected!

POP_TarjanPar(s ∈ Step)
  dfs.pop()
  ll ← llstack.pop(s.pos)
  if ll = s.pos then
    markdead(s)
  else
    p ← llstack.pop(dfs.top().pos)
    llstack.pushnontransient(min(p, ll))
    a ← uf.unite(s.src, dfs.top().src, s.acc)
    if a = F then
      stop ← ⊤
      report Accepting cycle detected!
    live.push(s.src)
```

Stratégie 11: Parallélisation de l'algorithme de Tarjan au moyen d'une structure d'union-find.

```
// Mark this SCC as Dead.
markdead(s : Step)
    uf.unite(s.src, alldead)
    livenum.remove(s.src)
    while livenum.size() > s.pos do
        q ← live.pop()
        livenum.remove(q)
```

Algorithme 6: Procédure permettant de marquer mort un ensemble d'états et de les supprimer localement.

```
parallel_main(str : Strategy)
    uf.makeset(alldead)
    EC(str, RANDOM) || ... || EC(str, RANDOM)
    Wait for all threads to finish
```

Algorithme 7: Procédure principale pour les tests de vacuité parallèles basés sur une structure d'union-find.

La méthode $\texttt{PUSH}_{TarjanPar}$ permet d'ajouter tous les nouveaux états directement dans la structure d'union-find (ligne 11). Cette opération insère aussi localement les états dans *livenum*. Cette insertion locale permet de gérer les conflits de parcours entre les threads et un état est dit vivant si et seulement s'il est présent localement. L'insertion dans l'union-find permet quant à elle d'agréger au maximum l'information de tous les threads pour constituer les composantes fortement connexes.

La méthode $\texttt{GET\_STATUS}_{TarjanPar}$ permet de connaître le statut d'un état en regardant d'abord localement s'il est vivant (test de la présence dans *livenum*) avant de tester si il est globalement marqué comme mort. L'ordre dans lesquels sont effectués ces tests permet de limiter la contention sur la structure d'union-find partagée. Ainsi, un état peut avoir été marqué comme mort globalement sans que cette information ne soit prise en compte par un autre thread. Ce dernier point est plus largement discuté à la section 9.4.2.

La méthode $\texttt{UPDATE}_{TarjanPar}$ est déclenchée à chaque détection d'une transition fermante, i.e dès qu'une partie de composante fortement connexe est détectée. La source et la destination de cette transition sont alors unies et l'ensemble de marques d'acceptation portée par cette transition est ajoutée à la classe contenant ces deux états dans l'union-find (ligne 26). Lors de cette union, le nouvel ensemble d'acceptation est retourné par la méthode `unite` et un contre-exemple est détecté si toutes les marques d'acceptation sont présentes. Dans ce cas, la variable *stop* est positionnée à $\top$ ce qui provoque l'arrêt des autres threads.

Lors d'un $\texttt{POP}_{TarjanPar}$, une union est faite entre l'état qui est en train d'être dépilé et son prédécesseur s'ils appartiennent à la même composante fortement connexe. Ainsi, lors de cette union la marque d'acceptation portée par la transition entre ces deux états (et stockée dans la pile *dfs*), est ajoutée à la classe contenant ces deux états dans l'union-find. De même que pour la méthode $\texttt{UPDATE}_{TarjanPar}$, le thread positionne *stop* de manière à arrêter le programme si l'ensemble des marques d'acceptation est retourné par la méthode `unite` puisque cela signifie qu'un cycle acceptant est détecté. Enfin si l'état qui est en train d'être dépilé est une racine,

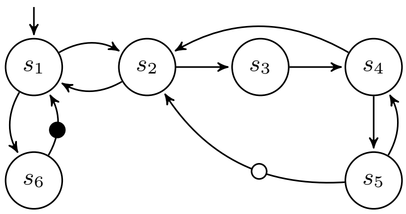


Figure 9.1 – Automate d'exemple.

Étape	Thread 1	Thread 2
1.	Visite s_1	
2.	Visite s_2	
3.	Visite s_3	
4.	Visite s_4	
5.	Détection (s_4,s_2)	
6.	Visite s_5	
7.	Détection (s_5,s_4)	
8.	Détection (s_5,s_2)	
9.		Visite s_1
10.		Visite s_6
11.		Détection (s_6,s_1)
12.		Visite s_2
13.		Détection (s_2,s_1)
14.	Fin visite s_5	
15.	Fin visite s_4	
16.	Fin visite s_3	
17.	Détection (s_2,s_1)	

Figure 9.2 – Exemple d'exécution avec deux threads.

une simple union avec la partition contenant *alldead* permet de marquer tous les états de la composante fortement connexe comme morts.

Avec cette stratégie, toutes les informations stables sont publiées dès qu'elles sont découvertes par un thread. Cela permet d'accélérer la propagation de l'information entre les différents threads. La structure d'union-find est donc toujours « à jour » au regard des explorations de tous les threads. Comme toutes les informations stables sont ajoutées dans l'union-find au moyen de la méthode `unite`, et que chaque thread teste toujours la valeur de retour de cette méthode, aucun cycle acceptant ne peut être manqué.

Note : Dans cet algorithme la compression de la pile des *lowlinks* proposée section 4.4 fonctionne puisqu'elle ne travaille que sur des informations locales. Notons néanmoins que cette pile ne stocke plus les ensembles de marques d'acceptation. De plus, un schéma de preuve pour cet algorithme est présenté en annexe B. Enfin la compatibilité avec le *Bit State Hashing* n'est pas immédiate puisqu'on ne peut appliquer les techniques vues au chapitre 5. Une solution serait de transférer tous les états dans la même partition que l'état artificiel *alldead* vers un ensemble utilisant cette technique.

9.2.2 Déroulement de l'algorithme

Exemple.
Cette section présente le déroulement de la stratégie 11 sur le TGBA de la figure 9.1. Cet automate est composé d'une unique composante fortement connexe et de deux marques d'acceptation ○ et ●. Pour plus de clarté nous ne considérons ici que deux threads dont l'entrelacement est présenté figure 9.2. Cet exemple met en avant la collaboration des threads car chaque marque d'acceptation est découverte par un thread différent.

La figure 9.3 présente l'évolution de la structure d'union-find pour cet entrelacement des

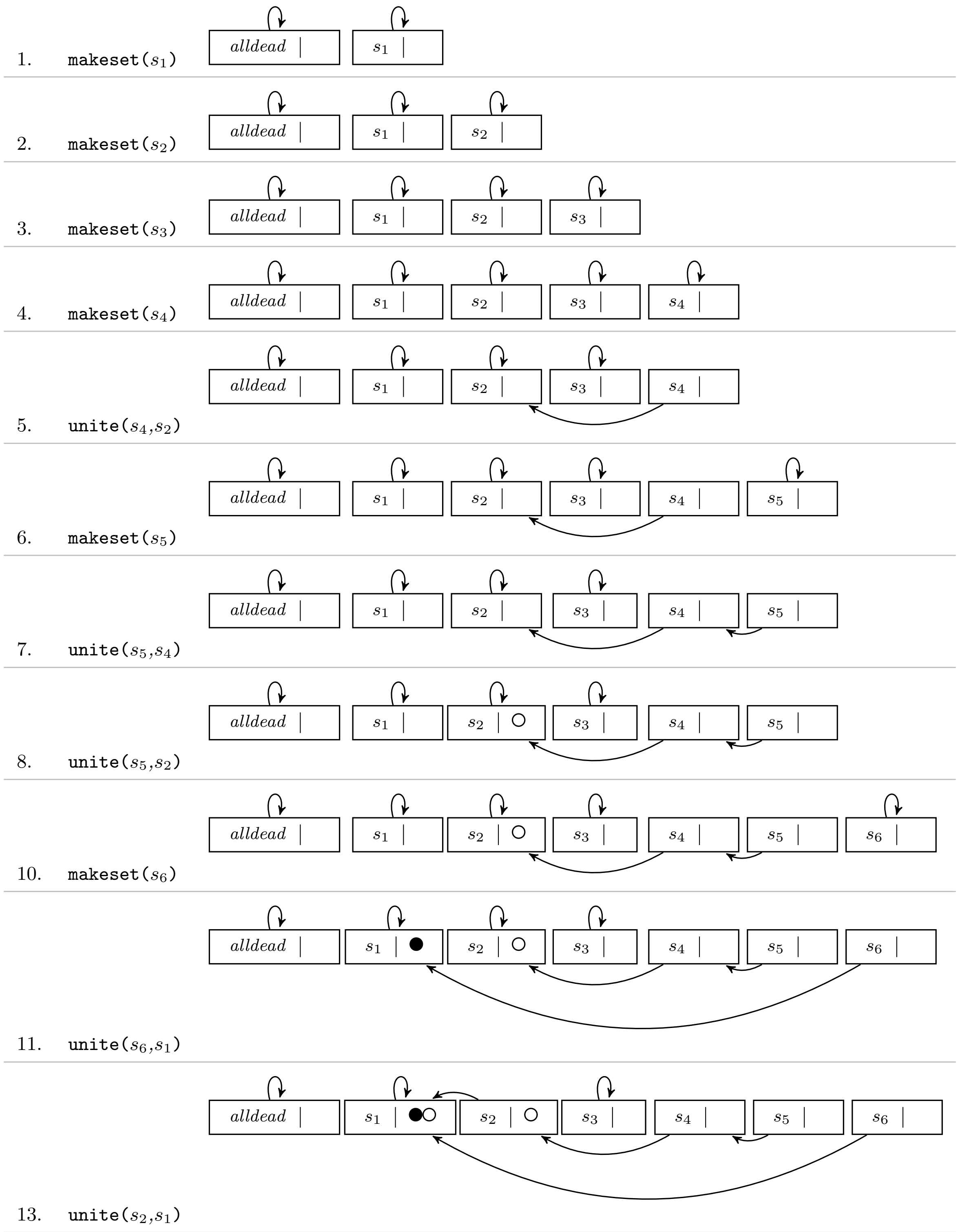


Figure 9.3 – Évolution de la structure d'union-find sur l'exemple de la figure 9.1 en suivant l'exécution présentée à la figure 9.2 avec la stratégie 11. (Seules les étapes modifiant la structure sont présentées. Les étapes 9 et 12 qui sont redondantes ne sont donc pas présentes.)

threads (seules les étapes affectant la structure d'union-find sont présentées). Les quatre premières étapes représentent la découverte des états s_1, s_2, s_3 et s_4 qui sont alors leurs propres représentants dans l'union-find. Toutes les classes crées sont associées à un ensemble d'acceptation vide puisqu'aucune opération `unite` n'a été effectuée. L'étape 5 présente la détection de la transition fermante (s_4, s_2) : les deux classes associées sont unies et s_2 devient le représentant de cette nouvelle classe. Comme la transition (s_4, s_2) ne porte aucune marque d'acceptation, l'ensemble des marques d'acceptation associé à l'état s_2 reste inchangé. L'étape 6 présente seulement l'insertion de l'état 5 dans l'union-find, tandis que les étapes 7 et 8 montrent la détection de deux transitions fermantes. Ces deux dernières étapes lient donc les états s_5, s_4 et s_2. Le nœud contenant l'état s_2 devient donc le représentant de cette classe et, comme la transition (s_5, s_2) porte la marque d'acceptation $\circ$, il est associé à l'ensemble $\{\circ\}$.

A l'étape 9 le deuxième thread reprend la main et explore le graphe avec un autre ordre de transitions. Comme l'état s_1 n'est « nouveau » que localement, l'insertion échoue puisqu'il est déjà présent dans l'union-find. En revanche la découverte de l'état s_6 à l'étape 10 conduit à son insertion dans l'union-find puisqu'il n'est pas déjà présent. La découverte de la transition fermante (s_6, s_1) avec la marque d'acceptation $\bullet$ conduit à l'union des états s_1 et s_6 dans l'union-find, et le nouveau représentant est associé à l'ensemble $\{\bullet\}$.

L'étape 12 n'est pas présentée dans la figure puisqu'elle ne correspond qu'à la découverte de l'état s_2 par le deuxième thread et qu'elle ne va pas modifier la structure d'union-find partagée. À l'étape 13, la transition fermante (s_2, s_1) est détectée. Comme cette transition fait le lien entre les classes $\{s_2, s_4, s_5\}$ et $\{s_1, s_6\}$ ces deux classes sont unies, et leurs marques d'acceptation aussi. Ainsi la nouvelle classe contient $\bullet$ et $\circ$ et le thread ayant découvert cette transition peut signaler l'existence d'un cycle acceptant et demander l'arrêt de tous les threads. Les étapes 14 à 17 ne seront utiles que pour la section suivante.

Il est intéressant de remarquer dans cet exemple que c'est l'étude d'une transition ne portant aucune marque d'acceptation qui permet de joindre les deux classes et de détecter ainsi un cycle acceptant. Dans cette stratégie, un cycle est détecté dès que toutes les transitions formant un cycle acceptant sont visitées. Dans cet exemple l'union-find n'a pas eu besoin d'apprendre que l'état s_3 fait aussi partie de la composante fortement connexe pour détecter un cycle acceptant.

9.3 Parallélisation de l'algorithme de Dijkstra

Dans la stratégie précédente chaque thread effectue $t + r$ opérations `unite` dans le pire cas, avec t le nombre de transitions dans les composantes fortement connexes et r le nombre de composantes fortement connexes.

Le nombre d'opérations `unite` a un impact direct sur la contention de la structure d'union-find puisque chacune de ces opérations doit utiliser des opérations atomiques pour retrouver les représentants des classes. Dans cette section nous présentons un algorithme parallèle, dans un parcours « à la Dijkstra », qui effectue moins d'opérations `unite`.

```
Structures supplémentaires :
  struct Step {src : Q,    succ : 2^Δ,
               acc : 2^F,  pos : int} // Refinement of Step of Algo 1

Variables Locales supplémentaires :
  live : stack of ⟨ Q ⟩
  livenum : map of Q ↦ ⟨p : int⟩
  rstack : pstack of ⟨ p : int, acc : 2^F ⟩ // Described Section 4.4

Variables Partagées supplémentaires :
  uf : union-find of ⟨ Q ∪ alldead, 2^F ⟩

PUSH_DijkstraPar(acc ∈ 2^F, q ∈ Q) → int
    uf.makeset(q)
    p ← livenum.size()
    livenum.insert(⟨ q, p ⟩)
    rstack.pushtransient(dfs.size())
    dfs.push(⟨ q, succ(q), acc, p ⟩)

GET_STATUS_DijkstraPar(q ∈ Q) → Status
    if livenum.contains(q) then
        return LIVE
    else if uf.contains(q) ∧ uf.sameset(q, alldead) then
        return DEAD
    else
        return UNKNOWN

UPDATE_DijkstraPar(acc ∈ 2^F, dst ∈ Q)
    dpos ← livenum.get(d)
    ⟨r, a⟩ ← rstack.pop(dfs.size() −1)
    a ← a ∪ acc
    while dpos < dfs[r].pos do
        ⟨r, la⟩ ← rstack.pop(r −1)
        a ← a ∪ dfs[r].acc ∪ la
        a ← unite(dst, dfs[r].src, a)
    rstack.pushnontransient(r, a)
    if a = F then
        stop ← ⊤
        report Accepting cycle detected!

POP_DijkstraPar(s ∈ Step)
    dfs.pop()
    if rstack.top(s.pos) = dfs.size() then
        rstack.pop(dfs.size())
        markdead(s) // Described Algorithm 6, page 135
    else
        live.push(s.src)
```

Stratégie 12: Parallélisation de l'algorithme de Dijkstra avec structure d'union-find.

9.3.1 Détails de l'algorithme

La stratégie 12 présente la parallélisation de l'algorithme de Dijkstra (stratégie 6, page 71) et une preuve de cet algorithme peut être trouvé en annexe B. Contrairement à l'algorithme de la section précédente où chaque thread publie les informations stables dès que possible, l'algorithme présenté ici ne publie des informations stables que lorsqu'une nouvelle racine potentielle est détectée. Cette publication « retardée » impose alors à chaque thread de mémoriser localement plus d'informations.

Comme pour l'algorithme précédent chaque thread exécute son propre test de vacuité et stocke localement l'information qui n'est pas stable. Ce test s'inspire de celui séquentiel basé sur l'algorithme de Dijkstra et utilise une pile des racines. Celle-ci stocke les positions dans DFS des racines des composantes fortement connexes et les marques d'acceptation associées.

L'insertion d'un état se fait grâce à la méthode $\texttt{PUSH}_{DijkstraPar}$ qui insère un état à la fois localement dans *livenum* et globalement dans l'union-find. Comme précédemment, cet algorithme intègre l'optimisation de Nuutila et Soisalon-Soininen et seuls les états vivants qui ne sont plus sur la pile *dfs* sont présents dans *live*. La méthode $\texttt{GET\_STATUS}_{DijkstraPar}$ vérifie si un état est mort en deux temps : d'abord localement dans *livenum*, puis globalement dans l'union-find.

Le point central de cette approche est la méthode $\texttt{UPDATE}_{DijkstraPar}$ déclenchée à chaque fois qu'une transition fermante est détectée. Si cette transition permet de découvrir une racine plus « ancienne » (pour l'ordre DFS du thread), i.e. lorsque la valeur du sommet de *rstack* change, tous les états entre l'ancienne racine et la nouvelle sont unis. Cette union prend aussi en compte les ensembles de marques d'acceptation et récupère ceux stockés dans l'union-find (ligne 30). À la fin de ces unions si toutes les marques d'acceptation sont présentes dans a, le thread peut positionner *stop* à $\top$ indiquant ainsi qu'un cycle acceptant a été détecté. Cependant, la détection d'une transition fermante ne conduit pas nécessairement à la détection d'une racine « plus ancienne » et dans ce cas aucune union n'est nécessaire puisque les états sont déjà dans la même partition : les marques d'acceptation découvertes sur la transition fermante sont uniquement stockées localement et la prise en compte de l'information qui a été calculée par les autres threads est retardée à la prochaine opération `unite`. Ce report permet à un thread de n'effectuer que n opérations `unite` pour une composante fortement connexe de taille n. Cette réduction a cependant un impact sur la détection des cycles acceptants qui ne peuvent être découverts que lorsque toutes les transitions formant le cycle acceptant ont été visitées par un thread.

Dans cette stratégie le partage de l'information stable se fait donc essentiellement lors d'un $\texttt{UPDATE}_{DijkstraPar}$ puisque la méthode $\texttt{POP}_{DijkstraPar}$ se charge seulement de marquer la racine d'une composante fortement connexe pour marquer globalement celle-ci comme morte.

9.3.2 Déroulement de l'algorithme

Exemple.
La figure 9.4 reprend l'exemple de la section précédente (avec le même ordre d'entrelacement des threads) et montre comment le report de la publication et de la prise en compte des informations stables impacte la détection des cycles acceptants et la structure d'union-find.

Les étapes 1 à 4 restent identiques puisqu'il ne s'agit que de la découverte et l'insertion des états s_1, s_2, s_3 et s_4. À l'étape 5, la détection de la transition fermante (s_4, s_2) conduit à l'union des états s_4, s_2 et s_3. Il s'agit là de la première différence avec l'algorithme précédent qui n'avait pas fait d'union avec s_3 pour détecter un contre-exemple. L'étape 6 présente simplement l'insertion de l'état s_5 dans la structure d'union-find et la détection de la transition fermante (s_5, s_4) à l'étape 7 construit la classe composée des états s_2, s_3, s_4 et s_5.

La détection de la transition fermante (s_5, s_2) à l'étape 8 ne modifie pas la structure d'union-find puisque les états s_2 et s_5 sont déjà dans la même partition. De plus, comme le sommet de la pile *rstack* référence déjà l'état s_2, aucune opération `unite` n'est réalisée : la marque d'acceptation ○ n'est pas ajoutée à la partition $\{s_2, s_3, s_4, s_5\}$. Elle est simplement stockée localement au sommet de la pile *rstack*.

Dès l'étape 9, le deuxième thread reprend la main, et l'étape 10 voit l'insertion de l'état s_6 dans la structure d'union-find. À l'étape 11 la transition fermante (s_6, s_1) est détectée et, comme précédemment, les classes des états s_1 et s_6 sont fusionnées. Lors de la découverte de la transition fermante (s_2, s_1) à l'étape 13 tous les états de cet automate sont unis au sein d'une même et unique classe. Contrairement à l'algorithme précédent la détection d'un cycle acceptant n'est pas possible à cette étape puisque la marque d'acceptation ○ n'a pas été publiée. Il faut attendre que le premier thread découvre lui même la transition (s_2, s_1) à l'étape 17 pour que la marque d'acceptation soit publiée et le cycle acceptant trouvé.

Discussion. Cet algorithme retarde donc la détection d'un cycle acceptant à l'étape 17 là où l'algorithme précédent le détecte dès l'étape 13. Cette détection « retardée » permet de limiter le nombre d'opérations `unite` : comme chacune de ces opérations utilise des opérations atomiques, leur réduction est importante. Dans la pratique, l'utilisation du swarming permet de compenser ce report et la minimisation du nombre d'opérations `unite` permet un meilleur passage à l'échelle (détails Chapitre 10).

9.4 Exploiter la structure d'union-find

Dans les deux sections précédentes nous avons vu que la structure d'union-find permet de partager les informations stables. Néanmoins, les différentes stratégies n'exploitent pas nécessairement toute l'information disponible. Dans cette section nous montrons comment mieux tirer parti de ces informations.

9.4.1 Combiner plusieurs algorithmes : stratégie *Mixed*

Le test de vacuité présenté section 9.2 favorise une détection collaborative des cycles acceptants tandis que le test présenté dans la section précédente permet de minimiser la contention sur la structure d'union-find partagé. Comme les informations partagées sont stables et ne dépendent pas de l'algorithme sous-jacent, l'idée de combiner ces algorithmes est naturelle. Ainsi, le nombre de thread disponible peut être divisé : une partie des threads va effectuer la stratégie *TarjanPar* tandis que l'autre partie va effectuer la stratégie *DijkstraPar*. L'algorithme 8 présente la procédure principale chargée d'instancier tous les threads avec la bonne stratégie. Lorsqu'il n'y a qu'un nombre impair de threads c'est la stratégie de *TarjanPar* qui est majoritaire.

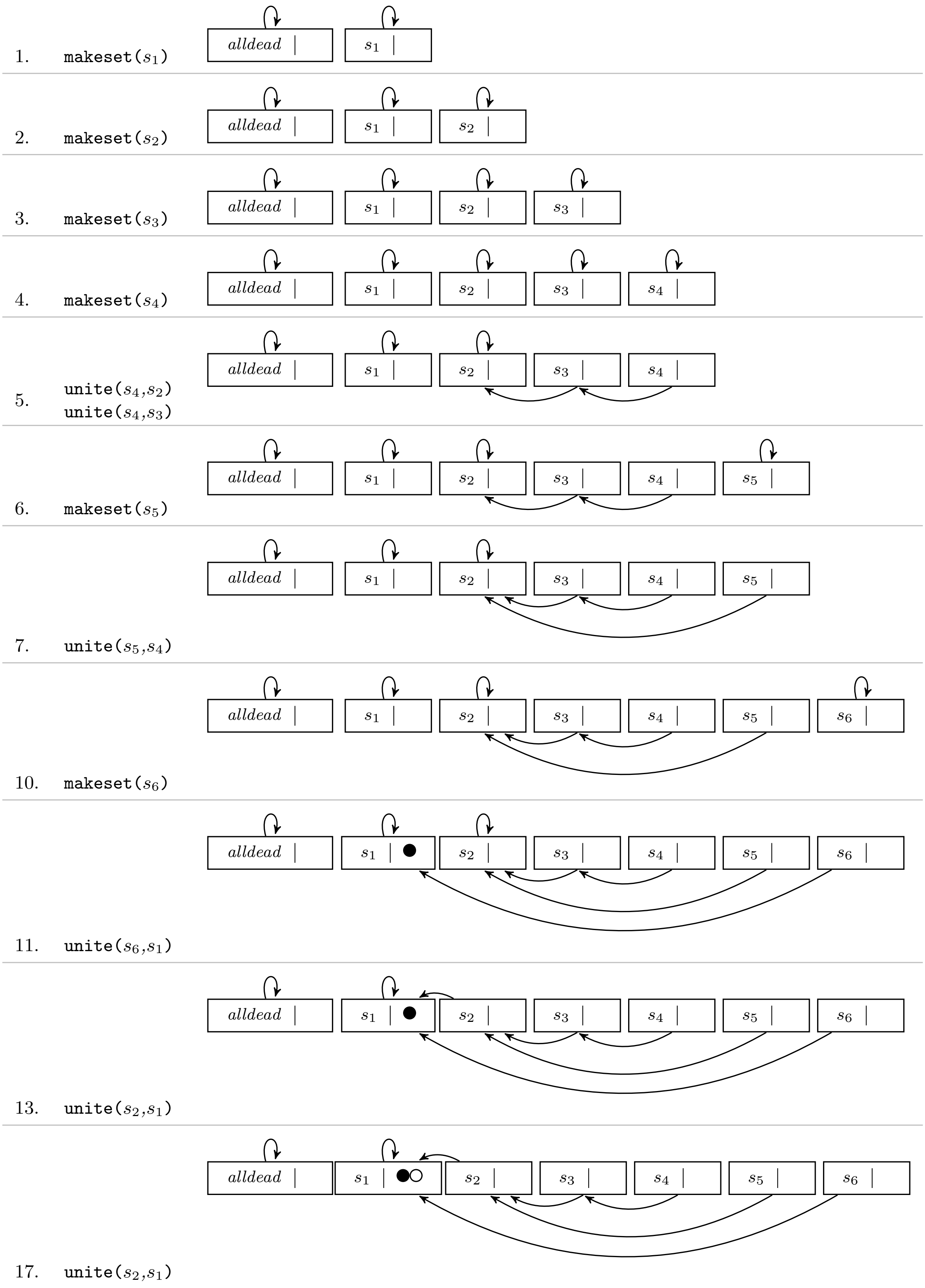


FIGURE 9.4 – Évolution de la structure d'union-find sur l'exemple de la figure 9.1 en suivant l'exécution présentée figure 9.2 avec la stratégie 12. (Seules les étapes modifiant la structure sont présentées.)

mixed_main(*nb_thread : int*)
    str1 $\leftarrow$ *dijkstraPar*
    str2 $\leftarrow$ *tarjanPar*
    $\mathtt{EC}_1(\mathit{str1},\ \textsc{random})\ ||\ \ldots ||\ \mathtt{EC}_{\lfloor \frac{n}{2} \rfloor}(\mathit{str1},\ \textsc{random})\ ||\ \mathtt{EC}_{1+\lfloor \frac{n}{2} \rfloor}(\mathit{str2},\ \textsc{random})\ ||\ \ldots ||$ $\mathtt{EC}_n(\mathit{str2},\ \textsc{random})$
    Wait for all threads to finish

Algorithme 8: Procédure principale pour la stratégie *Mixed*.

Note : Le partage de l'information stable permet donc de combiner plusieurs algorithmes en faisant complètement abstraction du test de vacuité utilisé. Ainsi les avantages de plusieurs algorithmes peuvent être couplés en portfolio pour obtenir de meilleures performances. Il est évident que la mise en place de cette stratégie n'est possible que parce que les stratégie *TarjanPar* et *DijkstraPar* utilisent des union-find de même nature et travaillent sur le même automate.

9.4.2 Limiter la redondance de calcul

Dans les deux algorithmes précédents, un thread peut ignorer qu'un état vient d'être marqué mort globalement. Il est donc inutile que ce thread continue à visiter les successeurs d'un tel état puisqu'ils ont été visités et qu'aucun cycle acceptant n'a été découvert. Un thread doit donc pouvoir détecter qu'un des états qu'il considère localement comme vivant vient d'être marqué globalement comme mort. Une première solution consiste à tester si l'état au sommet de la pile *dfs* est vivant à chaque itération (ligne 26 de l'algorithme générique, page 52). Cependant, cette technique augmente la contention sur la structure d'union-find et peut limiter le passage à l'échelle.

Une autre technique consiste ajouter un paramètre de retour à la fonction `unite`. Ainsi, cette opération renvoie, en plus d'un ensemble de marques d'acceptation, un booléen signalant si l'un des deux états est mort. Calculer cette information n'est pas coûteux car une opération `unite` calcule les représentants des deux éléments et la partition contenant *alldead* a toujours *alldead* comme représentant. Lorsqu'un état est signalé comme étant mort, le thread peut dépiler tous les états de la pile *dfs* qui sont eux aussi morts et mettre à jour la pile *live* en conséquence. Nous appelons cette technique FASTBACKTRACK puisqu'elle permet de remonter prématurément (sans avoir visité tous les successeurs) la pile *dfs*. Ainsi dès qu'une opération `unite` est réalisée par un des threads, ce dernier peut effectuer une opération FASTBACKTRACK s'il apparaît que les états considérés sont déjà morts.

Discussion. Dans ces algorithmes le statut d'un état est détecté en deux temps : d'abord localement puis globalement. Cet ordre a été choisi pour minimiser la contention sur la structure d'union-find. Néanmoins, on pourrait d'abord tester globalement si un état est mort. Dans ce cas, la contention sur la structure d'union-find serait plus forte mais les threads auraient un vision en « temps réel » du statut d'un état ce qui peut éviter d'effectuer du calcul redondant.

9.5 Conclusion

Ce chapitre a introduit deux nouveaux tests de vacuité parallèles (ainsi que leur combinaison) qui n'ont besoin ni de procédures de réparations ni de procédures de synchronisations. À notre connaissance, ces algorithmes sont les premiers supportant des automates généralisés et les premiers ayant une classe R_1–L_2 (dans la classification présentée au chapitre précédent). Cette classe est obtenue en étendant une structure d'union-find qui peut facilement être construite en utilisant des structures de données lock-free.

La stratégie *TarjanPar* détecte un cycle acceptant dès que l'ensemble des transitions le constituant ont été visitées par tous les threads. Cette détection « au plus tôt » a cependant un coût puisqu'elle augmente la contention sur la structure d'union-find partagée. Cette contention peut être réduite si l'on accepte une détection plus tardive et une consommation mémoire plus importante (stockage local des marques d'acceptation). La stratégie *DijkstraPar* stocke ainsi localement les marques d'acceptation qui ont été rencontrées et ne les publie que lors de la détection d'une nouvelle racine potentielle. Cette stratégie pourrait être raffinée pour publier systématiquement les marques d'acceptation lorsque l'ensemble d'acceptation associé à une racine est modifié.

Les deux stratégies évoquées ci-dessus ont des forces et des faiblesses qui peuvent être combinées comme le montre la stratégie *Mixed*. Cette combinaison d'algorithmes repose essentiellement sur le partage d'une information stable qui est indépendante de l'algorithme utilisé. Ainsi les algorithmes peuvent adapter leurs parcours et détecter plus rapidement les cycles acceptants.

On peut remarquer qu'une information non utilisée est partagée dans la structure d'union-find : le fait qu'un état soit en cours de traitement. En effet, dès qu'un état est découvert il est inséré dans l'union-find. Tant que cet état n'est pas dans la même partition que *alldead*, cela signifie que l'état est en train d'être traité par un thread. Cette information pourrait être utilisée pour restreindre le parcours des autres threads et ainsi accélérer les tests de vacuité.

Chapitre 10

Comparaison des algorithmes parallèles

You won't get sued for anticompetitive behavior.

Linus Torvalds

Sommaire

Dans les chapitres précédents nous nous sommes intéressés à la parallélisation des procédures de vérification pour les automates forts et/ou généralisés. Tout d'abord, nous avons proposé une décomposition de l'automate de la propriété qui tire parti des mélanges de forces dans automates. Ensuite, nous avons mis en place des tests de vacuité parallèles supportant les automates généralisés. Ce chapitre vise à étudier les performances de ces approches.

10.1 Évaluation de la décomposition des automates multi-forces

Au chapitre 7, nous avons proposé de paralléliser l'approche par automates pour le *model checking* afin d'améliorer la vérification de propriétés qui se traduisent par des automates multi-forces. Pour cela, l'automate de la propriété ($\mathcal{A}_{\neg\varphi}$) est décomposé en trois automates :

- $\mathcal{A}_T$: l'automate intrinsèquement terminal. Il capture uniquement les comportements acceptés par les composantes intrinsèquement terminales de $\mathcal{A}_{\neg\varphi}$;
- $\mathcal{A}_W$: l'automate intrinsèquement faible. Il capture uniquement les comportements acceptés par les composantes intrinsèquement faibles de $\mathcal{A}_{\neg\varphi}$;
- $\mathcal{A}_S$: l'automate fort. Il capture uniquement les comportements acceptés par les composantes fortes de $\mathcal{A}_{\neg\varphi}$.

Trois produits synchronisés sont ensuite réalisés. Il suffit alors de lancer trois tests de vacuité spécialisés en parallèle jusqu'à ce qu'un contre-exemple soit détecté ou que le dernier test de vacuité termine. Nous notons :

- $\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$: le produit synchronisé original ;
- $\mathcal{A}_T \otimes \mathcal{A}_{\mathcal{K}}$: le produit synchronisé avec l'automate terminal issu de la décomposition ;
- $\mathcal{A}_W \otimes \mathcal{A}_{\mathcal{K}}$: le produit synchronisé avec l'automate faible issu de la décomposition ;
- $\mathcal{A}_S \otimes \mathcal{A}_{\mathcal{K}}$: le produit synchronisé avec l'automate fort issu de la décomposition.

Dans cette section, nous nous intéressons à l'impact de cette décomposition aussi bien sur les procédures de vérification que sur les produits synchronisés. L'étude du jeu de test proposé au chapitre 6 montre que plus de deux tiers des formules qui le composent se traduisent en un automate multi-forces. La table 10.1 résume pour chaque modèle le nombre d'automates multi-forces générés et les regroupe en fonction de certaines de leurs caractéristiques :

- Type SW : l'automate est composé de composantes fortes et de composantes intrinsèquement faibles mais ne possède pas de composantes intrinsèquement terminales ;
- Type ST : l'automate est composé que de composantes fortes et intrinsèquement terminales mais ne possède pas de composantes intrinsèquement faibles ;
- Type SWT : l'automate est composé que de composantes fortes, intrinsèquement faibles et intrinsèquement terminales ;

Modèle	Type SW ✓	Type SW ×	Type ST ✓	Type ST ×	Type SWT ✓	Type SWT ×
adding.4	60	146	8	15	7	43
bridge.3	55	116	17	14	22	60
brp.4	15	18	1	0	3	7
collision.4	4	8	1	1	6	3
cyclic_scheduler.3	60	48	9	23	52	75
elevator2.3	125	180	14	12	30	50
elevator.4	14	21	2	1	2	7
exit.3	37	71	23	13	32	56
leader-election.3	75	24	15	37	88	240
production-cell.3	121	135	6	3	20	55
Total	566	767	96	119	262	596

TABLE 10.1 – Nombre de formules de chaque type par modèle. Nous distinguons pour chaque type les formules vérifiées (✓) de celles qui ne le sont pas (×).

Note : Ce jeu de test n'est composé que de formules qui se traduisent en automates forts. Nous ne considérons donc pas ici les formules produisant des automates ayant à la fois des composantes intrinsèquement faibles et intrinsèquement terminales. Néanmoins la décomposition y est entièrement applicable.

L'étude de la table 10.1 montre que les trois types d'automates multi-forces sont représentés dans ce jeu de test. On remarque que les automates les plus représentés sont ceux de Type 1, c'est-à-dire ceux pour lesquels la détection d'un contre-exemple ne peut pas être faite de manière prématurée. De plus, on voit que chaque type est représenté pour chaque modèle. Ce jeu de test est donc suffisamment représentatif pour évaluer la décomposition détaillée au chapitre 7. Dans cette section nous nous intéresserons seulement aux 2 406 formules produisant des automates multi-forces (924 générant des produits vides, 1 482 générant des produits non-vides).

La décomposition d'un automate multi-force permet d'obtenir trois automates qui sont plus petits en terme d'états, de transitions et de composantes fortement connexes. Des opérations de simplification peuvent ensuite être appliquées sur chaque automate décomposé puisqu'ils ont moins de contraintes que l'automate original. La table 10.2 présente à la fois l'impact de cette décomposition mais aussi celui des opérations de simplification. On remarque ainsi que :

- tous les automates décomposés sont strictement plus petits que les automates originaux ;
- comparés aux automates originaux, les automates terminaux ont en moyenne 20% d'états en moins, 11% de transitions en moins et 25 % de composantes fortement connexes en moins ;
- comparés aux automates originaux, les automates faibles ont en moyenne 29% d'états en moins, 40% de transitions en moins et 34 % de composantes fortement connexes en moins ;
- les automates forts sont ceux qui offrent la plus grosse réduction par rapport à l'automate original puisqu'ils possèdent moitié moins d'états et de transitions et qu'ils ont trois fois moins de composantes fortement connexes ;
- les opérations de simplification permettent de réduire le nombre d'états de chaque automate décomposé d'en moyenne 20%. De même, on observe une réduction du nombre de transitions allant de 20% à 35% et une réduction du nombre de composantes fortement connexes pouvant aller jusqu'à 35% pour les automates forts.

Automate	Nombre d'états			Nombre de transitions			Nombre de SCC		
	min	avg	max	min	avg	max	min	avg	max
$\mathcal{A}_{\neg\varphi}$	2	15	143	4	103	1 673	2	12	123
$\mathcal{A}_T$	2	12	88	2	91	876	2	9	57
$\mathcal{A}_W$	1	11	99	1	65	1 312	1	8	86
$\mathcal{A}_S$	1	7	81	2	51	958	1	4	61
$\mathcal{A}_T$ (+simplif.)	2	8	88	2	60	795	2	6	57
$\mathcal{A}_W$ (+simplif.)	1	9	82	1	47	1 312	1	7	70
$\mathcal{A}_S$ (+simplif.)	1	6	79	2	42	958	1	4	61

TABLE 10.2 – Impact de la décomposition sur l'automate de la propriété avec ou sans simplifications.

Ainsi, après les opérations de simplification, on note que chaque automate décomposé est en moyenne deux fois plus petit en nombre d'états, de transitions, et de composantes fortement connexes, que l'automate original. L'analyse des performances des tests de vacuité sur la décomposition (avec simplification) ne peut être faite sans avoir comparé la structure de l'espace d'état original avec les structures des espaces d'états décomposés. La table 10.3 montre cet impact sur les 941 produits vides du jeu de test.

	Nombre d'états min	avg	max	Nombre de transitions min	avg	max
$\mathcal{A}_{\neg\varphi} \otimes \mathcal{A}_{\mathcal{K}}$	110 931	2 923 870	44 071 003	451 852	9 642 553	122 470 050
$\mathcal{A}_T \otimes \mathcal{A}_{\mathcal{K}}$	1	387 718	41 465 547	0	1 220 568	113 148 822
$\mathcal{A}_W \otimes \mathcal{A}_{\mathcal{K}}$	1	1 796 544	42 706 623	0	5 753 903	121 400 696
$\mathcal{A}_S \otimes \mathcal{A}_{\mathcal{K}}$	1	1 686 822	42 829 923	0	5 535 723	121 400 696

Table 10.3 – Comparaison des caractéristiques des produits synchronisés sur 941 formules générant des produits synchronisés vides.

On constate tout d'abord que certains produits synchronisés décomposés ne possèdent qu'un état et n'ont pas de transitions. L'existence de tels produits montre que certains comportements capturés par les composantes fortement connexes faibles, fortes ou terminales ne seront jamais synchronisés. L'extraction de ces comportements au sein d'un automate permet alors de réduire les contraintes sur les autres automates pour les rendre plus petit ou plus déterministes : cela permet de réduire la taille du produit synchronisé et d'améliorer la vérification [9].

On remarque aussi que les produits synchronisés issus des automates intrinsèquement faibles ou forts ont en moyenne 40% d'états et de transitions de moins que le produit synchronisé original. Les gains les plus importants sont obtenus par les produits synchronisés issus des automates intrinsèquement terminaux puisque l'on observe une réduction de 86% du nombre d'états et de presque 90% du nombre de transitions. Chaque produit synchronisé est donc plus petit que le produit synchronisé original : cela aide à combattre l'explosion combinatoire.

Nous avons vu tout au long de ce manuscrit qu'il existe de nombreux tests de vacuité pour les automates forts tandis qu'il n'en existe qu'un pour les automates faibles et un pour les automates terminaux [1]. Lorsque l'on teste la vacuité de l'automate fort issu de la décomposition, on peut donc utiliser n'importe quel algorithme supportant de tels automates. Nous avons donc lancé des expérimentations avec tous les algorithmes présentés au chapitre 6.

Algorithme	Temps cumulé (924 produits vides) sans décomp.	avec décomp.	gain (%)	Temps cumulé (1482 produits non vides) sans décomp.	avec décomp.	gain (%)
`ndfs`	66 290 893	56 294 968	15	93 187 250	30 356 224	67
`tec`	64 761 041	55 671 158	14	80 617 945	25 818 980	68
`dec`	65 067 602	54 667 022	16	78 007 552	17 148 276	78
`tuf`	61 877 802	52 985 030	14	78 587 919	19 067 879	76
`duf`	63 181 423	53 555 449	15	77 204 985	17 301 099	78

Table 10.4 – Temps cumulé en millisecondes pour chaque algorithme et sa version décomposée. Le gain obtenu (en pourcentage, arrondi à l'unité) est aussi présenté.

1. Nous ne considérons dans cette section que les tests de vacuité basés sur les DFS.

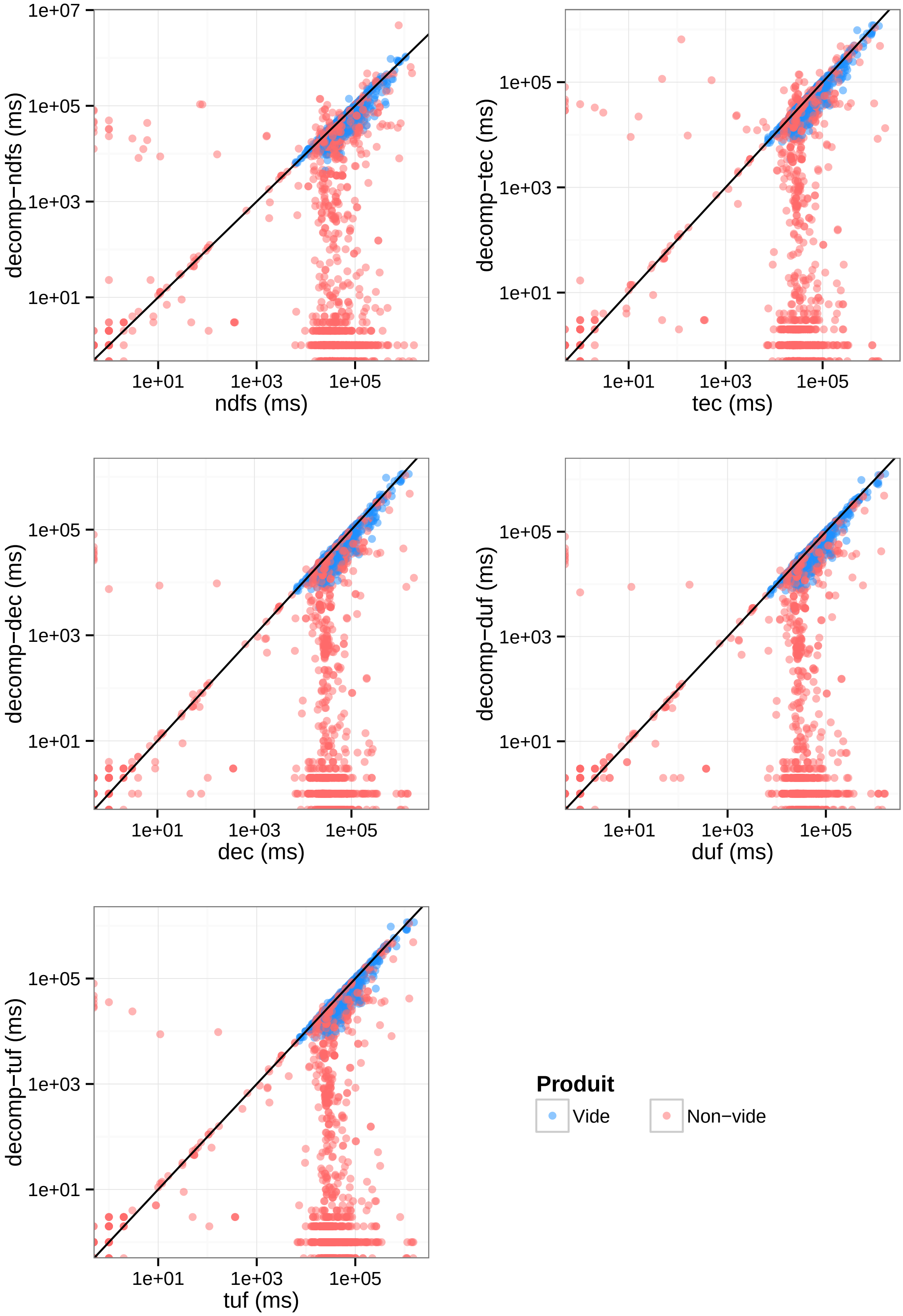


Figure 10.1 – Temps d'exécutions avec ou sans décomposition sur l'ensemble du jeu de test.

La technique de la décomposition produits trois automates plus petits à vérifier en parallèle. On s'attend donc à avoir des temps d'exécutions plus faible. L'évaluation présentée ici cherche à mesurer ce gain. La table 10.4 et la figure 10.1 présentent les résultats de cette évaluation. On constate tout d'abord une réduction de 15% du temps de calcul pour les formules vérifiées. Pour les formules violées, cette réduction oscille entre 65% et 75%. Pour ces dernières formules il est cependant nécessaire de minimiser cet impact : les opérations de simplification peuvent avoir modifié l'ordre des transitions et permettre une détection plus rapide des contre-exemples.

De plus, une analyse fine des résultats montre que :

- en moyenne, pour les produits vides, le test de vacuité fort prend le plus de temps dans 52% des cas, tandis que dans 48% des cas c'est le test de vacuité faible. On remarque que le test de vacuité terminal n'est jamais le plus long : cela peut s'expliquer par la taille des produits qu'il manipule (qui sont beaucoup plus petits) et l'utilisation d'un algorithme plus simple ;
- en moyenne, pour les produits vides, le test de vacuité fort est le plus rapide dans 12 % des cas, le faible dans 51% des cas tandis que le terminal l'est dans 37% des cas.

Dans cette section nous avons donc montré que la décomposition de l'automate de la propriété permet une réduction significative du temps de vérification aussi bien pour les formules vérifiées que violées. De plus, nous avons vu que les opérations de simplification permettent de réduire considérablement la taille de chaque automate décomposé. Enfin, nous avons vu que, en moyenne, la partie faible constitue la partie la plus longue à vérifier.

10.2 Évaluation des tests de vacuité parallèles

Dans la section précédente nous avons vu que la parallélisation des tests de vacuité permet de réduire le temps de vérification de manière significative. L'objectif de cette section est d'évaluer les tests de vacuité présentés au chapitre 9.

10.2.1 Analyse sur le jeu de tests

Dans cette section, nous évaluons les différents tests de vacuité parallèles présentés au chapitre 9 sur le jeu de test élaboré au chapitre 6.1. Nous distinguons trois tests de vacuité parallèles :

1. `tarjanpar` : présenté en stratégie 11 et qui repose sur l'algorithme de calcul de composantes fortement connexe de Tarjan ;
2. `dijkstrapar` : présenté en stratégie 12 et qui repose sur l'algorithme de calcul de composantes fortement connexe de Dijkstra ;
3. `mixed` : présenté à la section 9.4.1 et qui combine les deux algorithmes précédents.

Les figures 10.2, 10.3 et 10.4 montrent les performances de ces algorithmes sur le jeu de test présenté à la section 6.1. La figure 10.2 présente les performances obtenues en utilisant 12 threads au lieu d'un seul sur l'ensemble du jeu de tests : la ligne noire représente un temps identique, les lignes grises représentent l'accélération optimale (divisée par 12) ou le pire cas (multipliée par 12). On peut ainsi constater l'effet du swarming qui permet trouver des contre-exemples rapidement et dépasser l'accélération optimale.

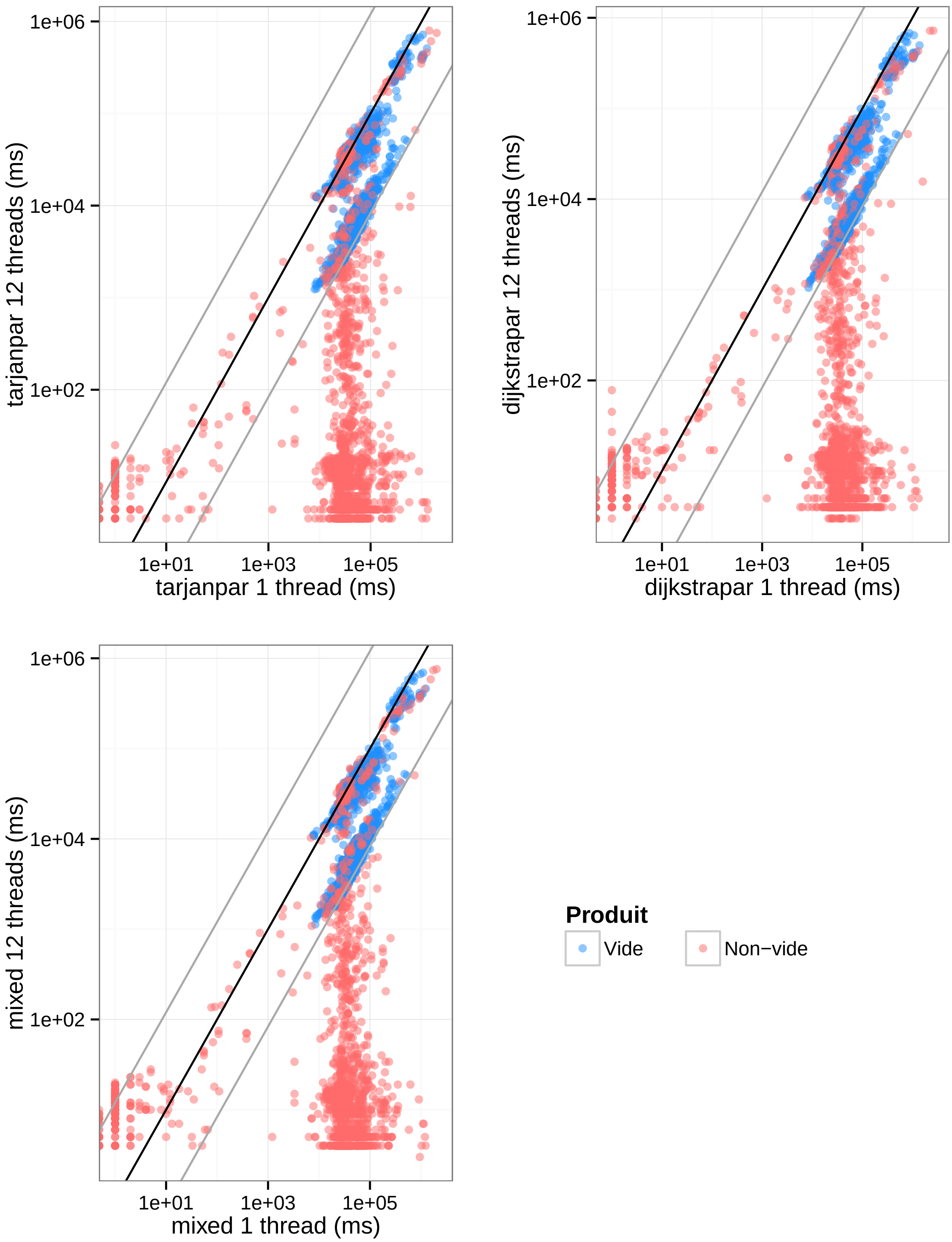


Figure 10.2 – Accélération des trois tests de vacuité en utilisant 12 threads.

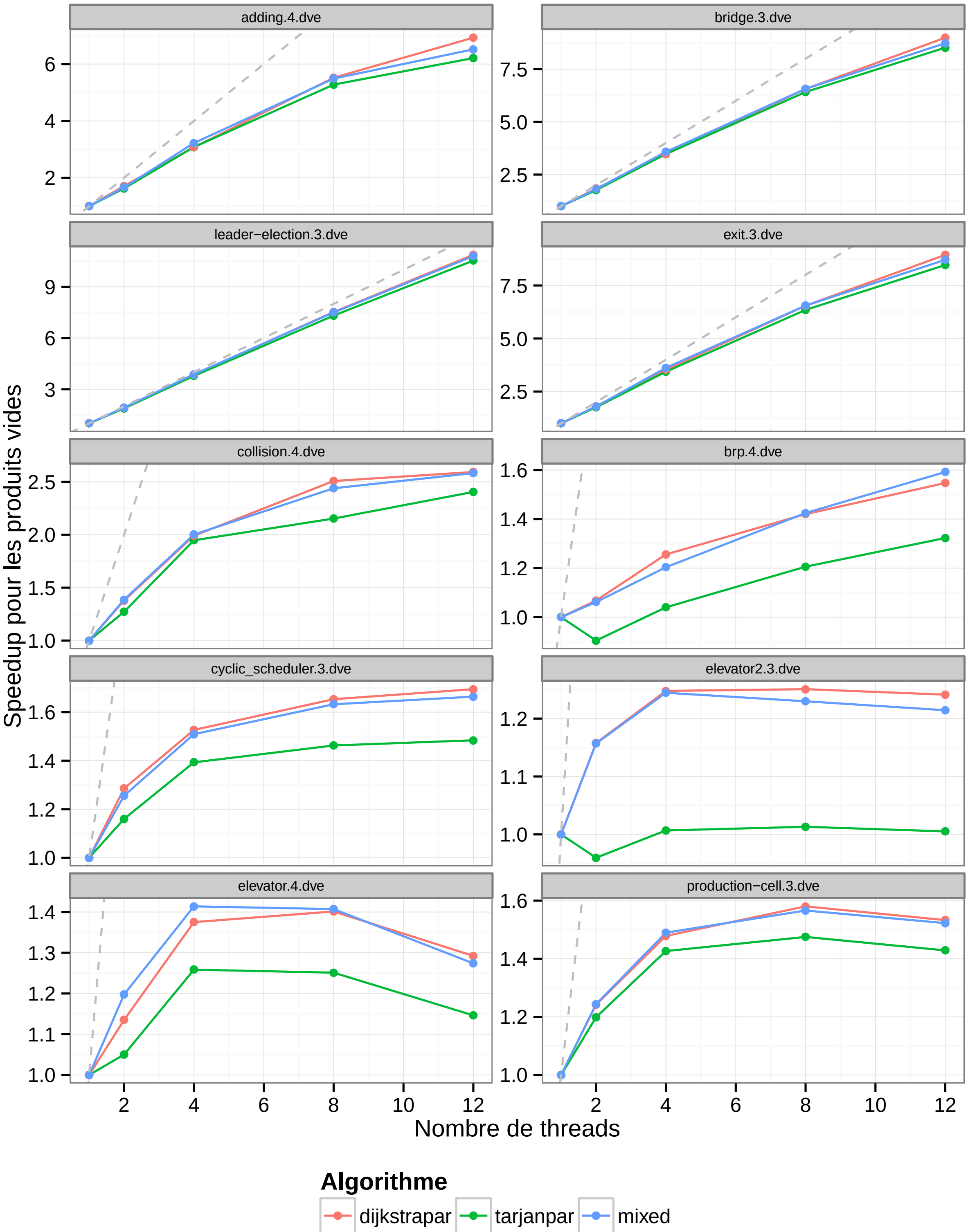


Figure 10.3 – Accélération des trois tests de vacuité sur les 1562 produits vides.

Dans les figures 10.3 et 10.4, nous représentons, pour chaque algorithme, l'accélération obtenue en utilisant deux, quatre, huit et douze threads. La ligne en pointillée représente l'accélération optimale (à savoir le temps d'exécution pour un thread divisé par le nombre de thread utilisé). Afin de mieux visualiser cette accélération l'échelle sur l'axe des ordonnées varie en fonction des modèles.

Si l'on regarde les produits pour lesquels il n'existe pas de cycle acceptant (cf. figure 10.3) on constate que les performances varient significativement en fonction des modèles. Ainsi, pour les modèles *adding.4*, *bridge.3*, *leader-election* et *exit.3* tous les tests de vacuité offrent une très bonne accélération. On note même que cette accélération est presque optimale pour les produits synchronisés issus du modèle *leader-election*. L'étude menée par Pelánek [65] montre que tous ces modèles ont des espaces d'états similaires à savoir de nombreuses composantes fortement connexes de petite taille et une structure qui a la forme d'un arbre. Ainsi, la table récapitulative de la page 91 montre que tous ces modèles sont de type (b) ou (d). Un même constat peut être fait en regardant la structure des produits synchronisés issus de ces modèles (cf. tables6.3 et 6.4, page 94). Tous partagent les mêmes caractéristiques : une petite hauteur DFS, un nombre très important d'états transients, des composantes fortement connexes de petite taille et un ratio transitions/états plutôt faible.

Toujours en considérant les produits vides, on remarque que les performances des différents tests de vacuité sont similaires sur les produits synchronisés issus des modèles *collision.4*, *brp.4* et *cyclic-scheduler.3*. Pour tous ces produits on note néanmoins une accélération nettement plus faible puisqu'elle peut descendre à 1.5 pour douze threads dans le pire cas. Néanmoins, cette accélération ne stagne pas : l'augmentation du nombre de threads peut donc encore améliorer les performances.

On constate à nouveau que les espaces d'état de ces modèles sont similaires (de type (a), détails page 91) puisqu'ils sont tous composés d'au moins une grosse composante fortement connexe composée de cycles longs. En regardant l'espace d'état des différents produits synchronisés, on remarque qu'ils ont tous une grosse taille DFS, très peu d'états transients, de nombreuses transitions, et un ratio transitions/états important. Notons néanmoins que pour les produits synchronisés issus des modèles *collision.4* et *cyclic-scheduler.3* les composantes fortement connexes sont de grosse taille tandis que pour le modèle *brp.4* elles sont de petite taille.

Les performances des algorithmes semblent donc plus liées au nombre de transitions et au ratio transitions/états qu'à la taille des composantes fortement connexes. Plus ce ratio est grand, plus il y a des opérations de recherche et d'union dans la structure d'union-find : cela augmente la contention sur cette structure. On remarque aussi que pour les produits synchronisés avec ces trois modèles les performances de la stratégie `tarjanpar` sont moins bonnes alors qu'elles sont quasiment identiques sur les quatre premiers modèles. Dans cette stratégie chaque thread effectue une union par transition qu'il visite. Comme le nombre de transitions est relativement important, la contention sur la structure d'union-find est augmentée.

Enfin, les performances des tests de vacuité sur les trois derniers modèles (*elevator2.3*, *elevator.4* et *production-cell*) sont plutôt mauvaises puisque l'augmentation du nombre de threads ne permet qu'une très faible amélioration du temps de vérification. De plus, plus le nombre de threads augmente, plus cette accélération tend à diminuer. L'analyse de ces modèles montrent qu'ils ont tous des caractéristiques identiques (ils sont tous de type (c)) à savoir : une grosse taille DFS, très peu voire pas d'états transients, et une grosse composante fortement connexe à la structure complexe. L'analyse des produits synchronisés qui en sont issus fait ressortir les mêmes caractéristiques. Comme pour les modèles *collision.4*, *brp.4* et *cyclic-scheduler.3* on remarque

que la contention sur la structure d'union-find joue un rôle déterminant sur les performances des tests de vacuité. Dans les cas où l'on utilise que deux threads on remarque même que le test de vacuité est pénalisé (pour deux modèles). Il s'agit de cas où les threads sont orientés vers les mêmes composantes fortement connexes et n'interfèrent.

Cette analyse semble montrer que les meilleures performances (pour les tests de vacuité parallèles basés sur un union-find) sont obtenues en présence de produits synchronisés composés de nombreuses composantes fortement connexes de petite tailles (organisées sous la forme d'un arbre) et ayant un faible ratio transitions/états. La section 10.2.2 justifie cette assertion.

La figure 10.4 montre l'accélération obtenue par les tests de vacuité sur les produits non vides. Tout d'abord on remarque que cette accélération est meilleure que pour les produits vides : cela est dû d'une part à l'utilisation du SWARMING et, d'autre part, au partage d'information dans la structure d'union-find. On constate deux catégories de produits synchronisés :

- ceux qui ont une accélération supérieure à l'accélération optimale : cela n'est possible que parce que la détection d'un contre-exemple ne nécessite pas de visiter l'intégralité de l'espace d'état du produit. Les produits pour lesquels on observe une telle accélération sont ceux qui ont une accélération presque optimale sur les produits vides ;
- ceux qui ont une accélération inférieure à l'accélération optimale. Ces produits sont pénalisés par la structure de l'espace d'état qui augmente la contention sur la structure d'union-find. Comme les contre-exemples sont détectés grâce à cette structure, plus la contention augmente, plus les performances diminuent.

De manière plus générale on remarque que les meilleures performances sur les produits vides sont obtenues en utilisant le test de vacuité `dijkstrapar`. En revanche, les plus mauvaises performances sont obtenues en utilisant l'algorithme `tarjanpar`. Ces différences sont essentiellement dues au nombre d'unions : comme expliqué dans le chapitre 9, dans la stratégie `tarjanpar` les threads effectuent une union par transition explorée, tandis que dans la stratégie `dijkstrapar` les threads effectuent une union par état exploré.

Enfin, on note que les performances de la stratégie `mixed` sont moyennes sur les produits vides tandis qu'elles tendent à être meilleures sur les produits non vides. Cette stratégie semble donc plus adaptée à la détection de contre-exemples qu'à la validation de propriétés.

10.2.2 Passage à l'échelle et contention

Dans la section précédente nous avons vu que la structure de l'espace d'état du produit semble impacter les performances des différents tests de vacuité. L'objectif de cette section est d'évaluer cet impact pour corroborer les intuitions alors émises.

Pour cela nous définissons trois modèles dont les espaces d'état reprennent les principales caractéristiques exhibées par l'étude de Pelánek [65]. Ces trois modèles sont composés de deux processus : le premier donnant la forme de l'espace d'état tandis que le second vient se greffer sur chaque nœud du premier processus pour construire une composante fortement connexe (un simple cycle de plusieurs états). Ces trois modèles sont présentés figure 10.5 et sont dénotés par :

tree : l'espace d'état forme un arbre de hauteur 20. Chaque état du premier processus est composé de trois successeurs et possède en plus une transition bouclant sur lui même. Le second processus forme un cycle composé de 10 états. Cet espace d'état est ainsi composé de nombreuses composantes fortement connexes de petite taille réparties sous la forme d'un arbre.

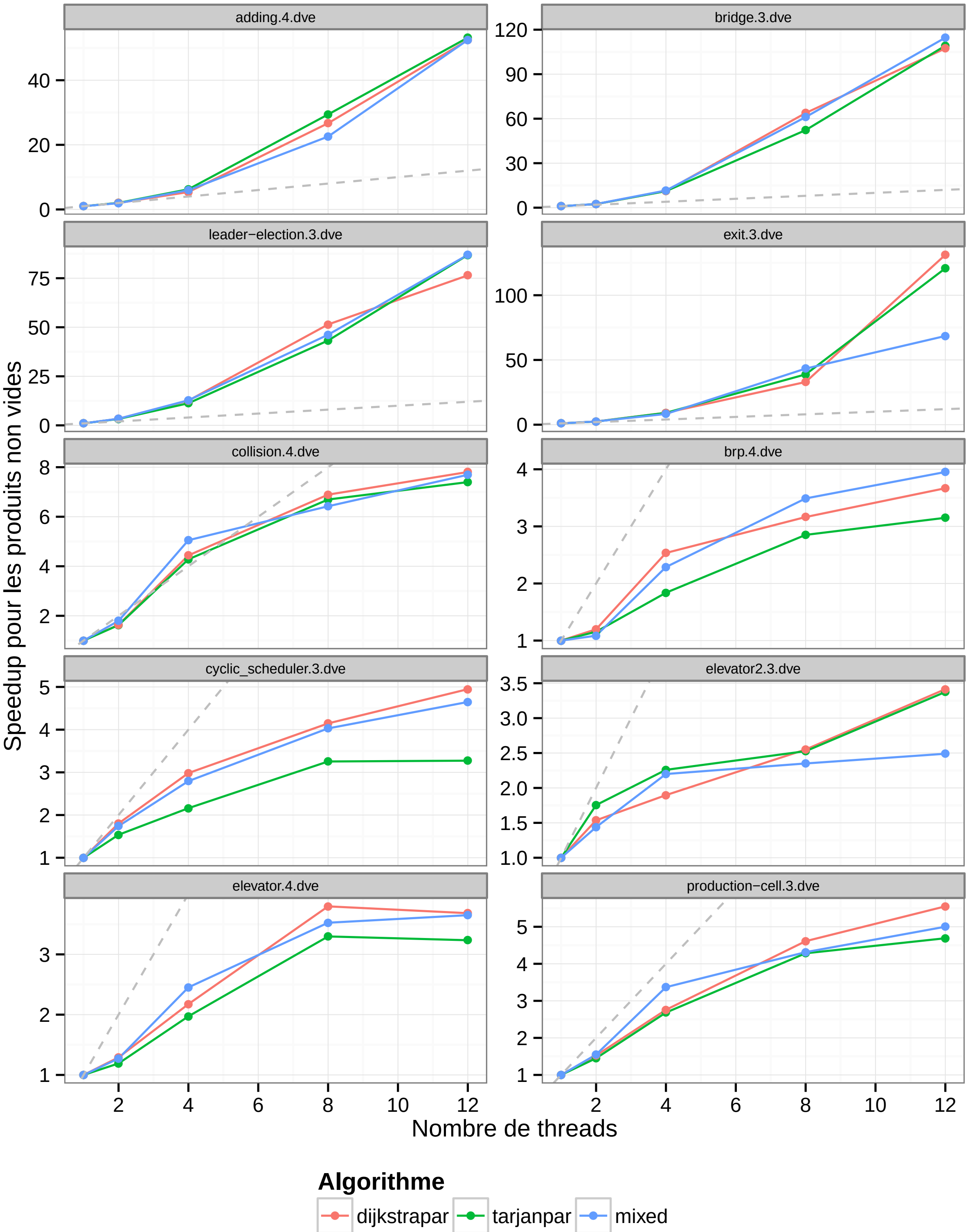
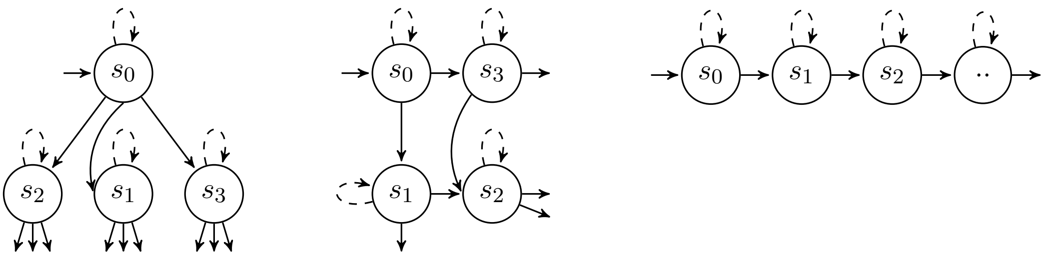


Figure 10.4 – Accélération des trois tests de vacuité sur les 1706 produits non vides.



(*tree*) (*lattice*) (*chain*)

Figure 10.5 – Trois nouveaux modèles pour tester la contention et le passage à l'échelle.

lattice : l'espace d'état forme un treillis de taille 1500 sur 200. Chaque état du premier processus est donc composé de deux successeurs plus une transition bouclant sur lui-même. Le second processus forme un cycle composé de 10 états. Cet espace d'état est ainsi composé de nombreuses composantes fortement connexes de petite taille réparties sous la forme d'une grille.

chain : l'espace d'état forme une chaîne de hauteur 110. Chaque état du premier processus est composé d'un successeur et d'une transition bouclant sur lui même. Le second processus forme un cycle de 1 000 états. Cet espace d'état est donc composé de grosses composantes fortement connexes qui s'enchaînent les unes au autres.

Notons que les valeurs des différents modèles n'ont pas été prises au hasard. Ces trois modèles génèrent en effet des espaces d'états dont la taille est proche : 10^6 pour *chain*, 3×10^6 pour *lattice* et 4×10^6 pour *tree*. Ensuite, l'analyse du jeu de tests à montré que la taille moyenne des composantes fortement connexes est de 10 lorsque le produit synchronisé est composé de nombreuses SCC. Cela justifie notre choix pour la taille des composantes pour les modèles *tree* et *lattice*. De même, les composantes de taille 1 000 pour le modèle *chain* correspondent à une estimation moyenne de la taille des composantes dans les produits synchronisés ayant de grosses composantes.

Dans cette section nous nous focalisons uniquement sur le test de vacuité `dijkstrapar` qui offre les meilleures performances (d'après la section précédente). Afin d'évaluer les performances ce test, nous vérifions le produit synchronisé entre les trois modèles présentés ci-dessus et la formule LTL « $\mathsf{G}\,\mathsf{F}\,P\_0.neverreached \wedge \mathsf{G}\,\mathsf{F}\ P\_1.neverreached$ ». Cette formule représente une propriété d'équité faible non vérifiée. Le produit résultant de chaque modèle est vide ce qui permet de comparer la contention et le passage à l'échelle sans être affecté par la détection d'un cycle acceptant. De plus, comme la formule se traduit en un TGBA déterministe et complet à un état, la structure du produit synchronisé est la même que la structure du modèle.

La figure 10.6 montre l'évaluation de ce test de vacuité sur les modèles *chain*, *tree* et *lattice*. On note tout d'abord les très bonnes performances obtenues sur le modèle *tree* puisque jusqu'à 14 threads l'accélération est presque optimale. À partir de 16 threads on observe que cette accélération diminue. Nous avons remarqué, durant ces expérimentations, que la mémoire utilisée à partir de 16 threads était proche de la mémoire disponible : cela fournit une explication quant à l'affaissement des performances. De plus, l'évaluation sur des arbres de taille plus petite ont montré que cet affaissement était retardé à l'utilisation de 18 ou 20 threads.

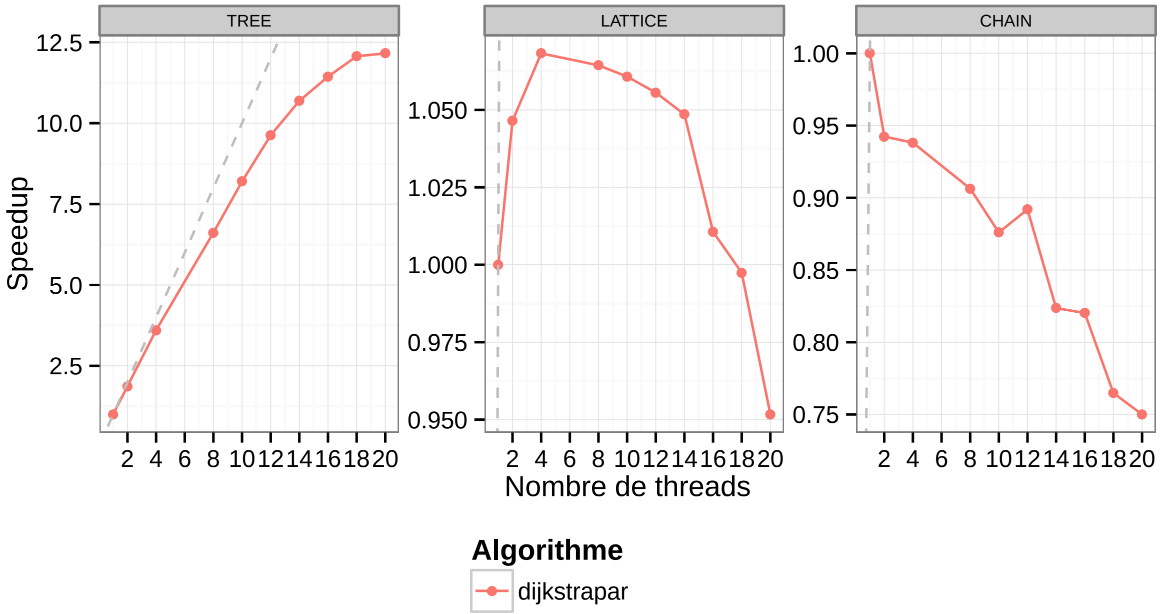


Figure 10.6 – Accélération sur chaque type d'espace d'état.

On remarque néanmoins que pour les deux autres modèles les gains sont nuls voire même négatifs puisque l'utilisation de plusieurs threads pénalise le temps de vérification. Cela s'explique simplement par l'étude de la structure de l'espace d'état du produit qui possède une unique composante fortement connexe « terminale », i.e. sans successeurs. Les threads évoluant à peu près de la même manière, tous arrivent sur cette composante en même temps et essayent de la traiter ce qui augmente la contention sur la structure d'union-find. Lors de la remontée le même problème survient de proche en proche.

Dans cette section nous avons vu que les hypothèses émises à la section précédente se sont révélées exactes : les tests de vacuité parallèles présentés dans ce manuscrit ont un meilleur passage à l'échelle lorsque l'espace d'état du produit forme un arbre. Il s'agit en effet du meilleur cas puisque chaque thread peut marquer certaines parties comme morte sans entrer en conflit avec les autres threads.

10.2.3 Comparaison avec les tests de vacuité parallèles existants

Intéressons-nous maintenant à la comparaison de ces nouveaux tests de vacuité avec ceux existants. Faute de temps, nous n'avons pas pu implémenter les principaux tests de vacuité parallèles pour effectuer une comparaison dans les mêmes conditions. Nous avons donc décidé de comparer les outils en mesurant l'accélération apportée par l'augmentation du nombre de threads. Les deux principaux *model checker* proposant des tests de vacuité parallèles sont : LTSmin [2] et Divine [3]. L'algorithme choisi pour LTSmin est le `cndfs` qui est reconnu comme le meilleur test de vacuité parallèle basé sur un NDFS. L'algorithme choisi pour Divine est `owcty` qui est basé sur un BFS et offre le meilleur compromis entre performances théoriques et pratiques [27].

2. http://fmt.cs.utwente.nl/tools/ltsmin/
3. http://divine.fi.muni.cz (version 2.4)

Note : Avant d'analyser les différents résultats il est important de noter que (1) nos formules sont traduites en utilisant LTL2TGBA tandis que LTSmin et Divine utilisent LTL2BA ; (2) nos tests de vacuité et ceux de Divine effectuent le produit synchronisé à la volée tandis que LTSmin repose sur un produit synchronisé pré-calulé [4], (3) notre implémentation constitue un prototype et n'intègre pas les multiples optimisations (caches, SWARMING par permutations, allocateur mémoire dédié, ...) que peuvent avoir les autres outils plus matures, et (4) les trois outils n'utilisent pas les mêmes fonctions de calcul des successeurs.

Afin d'évaluer ces trois outils, nous avons repris le jeu de test présenté à la section 6.1 et borné le temps d'exécution d'un test à une heure. Nous avons alors constaté que :

- Spot traite toutes les formules dans ce délais d'une heure ;
- Divine échoue à traiter 14 formules ;
- LTSmin échoue à traiter 787 formules sur les 3268 que compte notre jeu de test. L'étude de ces mauvaises performances montre que c'est la compilation de la bibliothèque représentant le produit synchronisé qui n'aboutit pas dans le temps imparti (plus de détails annexe A).

Dans cette section, nous restreignons le jeu de test au sous ensemble des couples (modèle, formule) pouvant être traité par tous les outils. Ce jeu de tests restreint est composée de 1 214 formules générant des produits vides et de 1 266 formules générant des produits non vides. De plus, nous ne nous intéressons ici qu'au temps pris par les tests de vacuité et ne considérons donc pas toutes les étapes préalables telles que la génération de la formule, sa minimisation ou la génération du produit synchronisé.

La figure 10.7 présente la comparaison des trois tests de vacuité sur le sous ensemble des formules pouvant être traitées à la fois par LTSmin, Spot et Divine dans un délai d'une heure. Pour chaque outil nous avons comparé les performances des tests de vacuité avec un thread et avec huit threads. Lorsqu'un seul thread est utilisé on remarque que l'algorithme `cndfs` est en moyenne trois fois plus rapides que `owcty` ou `dijkstrapar`. On remarque aussi que les tests de vacuité `owcty` et `dijkstrapar` ont des perfomances comparables même si `dijkstrapar` est en moyenne 5% plus lent. Lorsque l'on compare les performances en utilisant huit threads on constate que `cndfs` est toujours trois fois plus rapide que les autres algorithmes tandis que l'algorithme `owcty` est plus lent de 20% comparé à `dijkstrapar`. On note ainsi que l'augmentation du nombre de threads a permis à cet algorithme d'inverser le rapport de force.

Discussion. À la différence des autres outils, le notre n'intègre pas d'allocateur mémoire dédié. Ce-dernier peut avoir un impact très important sur les performances et une comparaison juste des outil forcerait l'utilisation d'un seul et même allocateur mémoire. Néanmoins, pour certains outils ces allocateurs sont profondément enfouis dans l'outil et difficilement modifiables.

Ces résultats permettent d'apprécier la maturité et le degré d'optimisation des outils. Néanmoins ils sont difficilement utilisables pour évaluer et comparer le passage à l'échelle des tests de vacuité. La figure 10.8 remédie à cela en présentant l'accélération obtenue pour chaque algorithme en faisant varier le nombre de threads de un à huit. Pour chaque modèle nous distinguons l'accélération obtenue sur les produits vides de celle obtenue sur les produits non vides.

4. LTSmin est lui aussi capable de générer le produit synchronisé à la volée, mais durant nos expérimentations nous avons détecté un bug dans la connexion avec LTL2BA. L'équipe de LTSmin nous a alors suggéré de pré-calculer le produit synchronisé pour mener nos expérimentations.

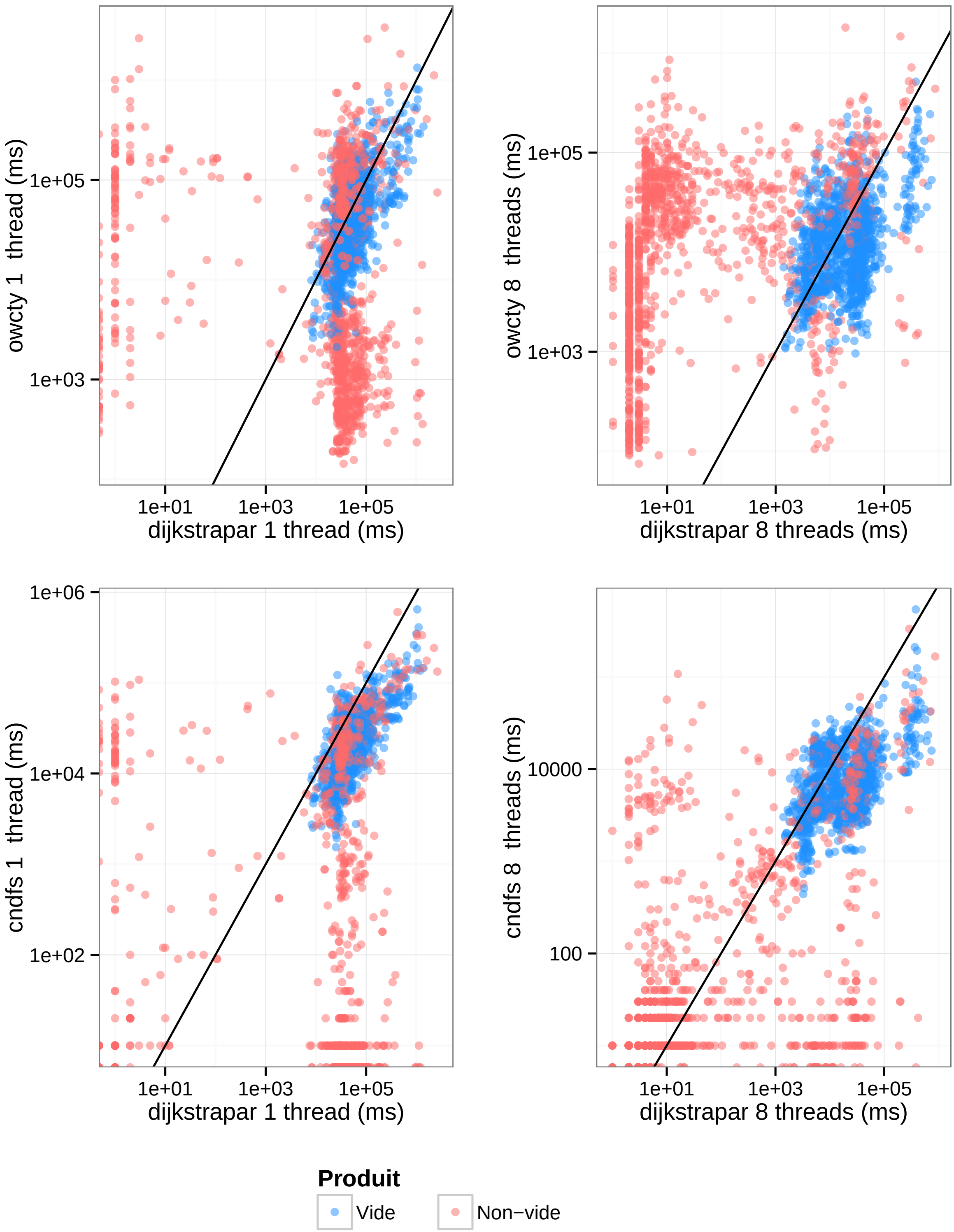


Figure 10.7 – Comparaison des performances brutes des différents outils.

Nous remarquons tout d'abord que dans la moitié des cas l'accélération obtenue par `dijks-`

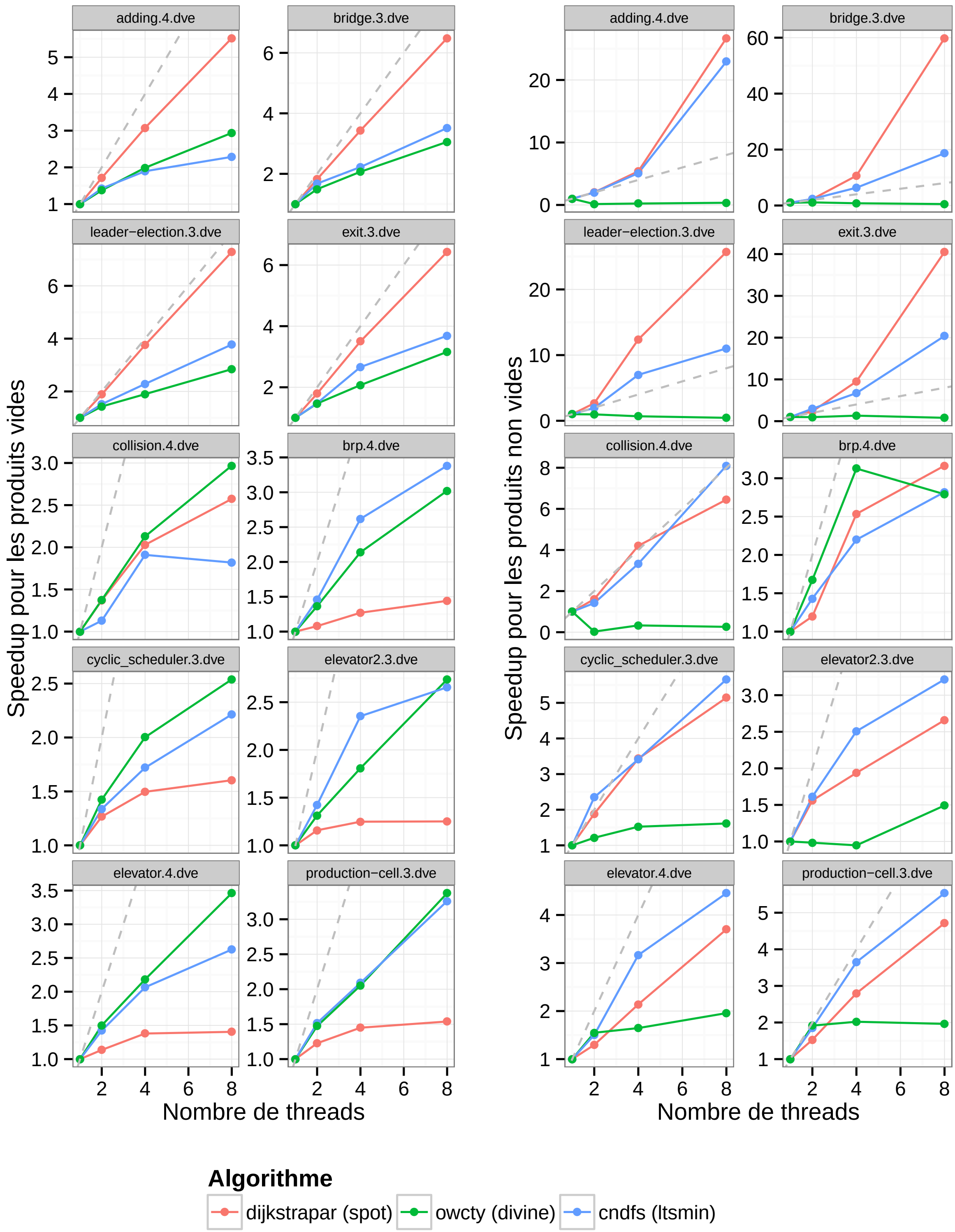


Figure 10.8 – Accélération pour chaque algorithme en fonction du nombre de threads.

`trapar` est supérieure à celle obtenue par `cndfs`. Pour les produits vides, on voit aussi que dans

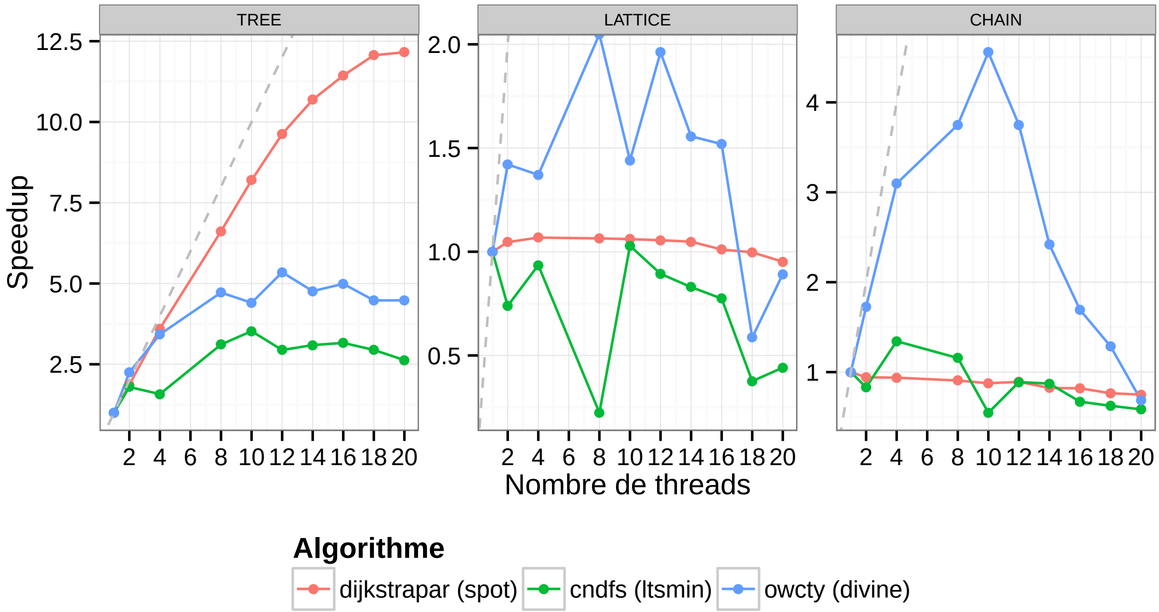


FIGURE 10.9 – Accélération pour chaque algorithme en fonction du nombre de threads sur les trois structure d'espace d'état du produit présenté à la section précédente.

la moitié des cas `cndfs` possède une accélération qui est meilleure que celle de `owcty`. Les différences les plus importantes se situent sur les produits non vides où le test de vacuité `owcty` possède une accélération quasi-nulle pour deux tiers des modèles. Par opposition, les deux autres tests de vacuité bénéficient d'une très bonne accélération grâce au SWARMING et à leur capacité à détecter les transitions fermantes.

La mise en relation de ces performances avec les tables 6.3 et 6.4 (pages 95 et 94) montre que ces trois algorithmes sont complémentaires :

- `dijkstrapar` offre les meilleures accélérations lorsque l'espace d'état du produit est composé de petites composantes fortement connexes organisées sous la forme d'un arbre. On voit ainsi que tous les produits synchronisés avec les modèles de type (a) possèdent une très bonne accélération ;
- `owcty` offre les meilleures accélérations lorsque les produits sont vides et que la structure de l'espace d'état est complexe. En effet, cet algorithme est basé sur un BFS et possède une bonne donc parallélisation sur les produits vides. Néanmoins, la propagation des marques d'acceptation est coûteuse en présence de cycles acceptants.
- `cndfs` offre les meilleures accélérations pour les produits non vides ayant des structures complexes. En effet, cet algorithme ne partage pas les marques d'acceptation et n'a donc pas les problèmes de contention que peut avoir `dijkstrapar`. De la même manière, on remarque que pour les produits vides le `cndfs` obtient de bien meilleures performances que `dijkstrapar` lorsque les composantes fortement connexes ont des structures complexes.

Afin de valider la pertinence de ces observations et d'évaluer le passage à l'échelle de ces trois algorithmes nous avons calculé leur accélération sur les trois produits (*chain*, *tree* et *lattice*) de la section précédente. Ces résultats sont présentés figure 10.9. Nous observons que :

- les modèles *lattice* et *chain* constituent des cas d'études compliqués pour les algorithmes à base de NDFS. En effet, pour le produit avec le modèle *chain* le SWARMING peut difficilement s'appliquer puisque le ratio états/transitions est faible. Pour le modèle *lattice*, tous les threads finissent par effectuer du travail redondant pour marquer des états comme morts.
- l'algorithme `owcty` possède une très bonne accélération pour le modèle *chain* jusqu'à 12 threads. Ensuite cette accélération s'écroule : comme l'espace d'état du produit n'est pas large, il arrive un point où il n'y a pas assez de traitement à effectuer pour tous les threads ;
- les algorithmes `owcty` et `cndfs` ont une accélération proche pour le modèle *tree* et que cette accélération est nettement inférieure à celle obtenue par `dijkstrapar`.

10.3 Conclusion

Dans ce chapitre nous avons évalué les parallélisations proposées dans ce manuscrit. Nous avons tout d'abord évalué l'impact de la décomposition sur les automates de la formule qui sont multi-forces. Pour ces automates, le gain lors du test de vacuité est d'en moyenne 15% pour les produits vides et 60% pour les produits non vides. Cette technique offre en plus l'avantage de réduire le nombre de propositions atomiques ce qui est un avantage pour les techniques d'ordre partiel.

Nous avons ensuite évalué les performances des tests de vacuité généralisés parallèles proposés au chapitre 9. L'analyse des performances a montré que les meilleurs résultats étaient obtenus par l'algorithme basé sur `dijkstrapar` qui limite la contention sur la structure d'union-find partagé. De plus, cet algorithme possède les meilleures accélération lorsque l'espace d'état est composé de petites composantes fortement connexes organisées sous la forme d'un arbre.

Enfin, nous avons mesuré l'accélération obtenue par les autres tests de vacuité sur le même jeu de tests. Ces résultats ont montré que les trois algorithmes sont complémentaires : `dijkstrapar` gère particulièrement bien les espaces d'état arborescents, `cndfs` est efficace pour la détection des cycles acceptants lorsque la structure de l'espace d'état est complexe et enfin `owcty` possède une bonne accélération pour les espaces d'état ayant des structures de composantes fortement connexes complexes.

Conclusion générale et perspectives

One more thing ...

Steves Jobs

Sommaire

Dans ce manuscrit, nous avons vu que les tests de vacuité basés sur le calcul des composantes fortement connexes permettent de traiter efficacement les automates généralisés. Nous avons alors proposé plusieurs techniques pour améliorer les performances de ces tests aussi bien dans un contexte séquentiel que dans un contexte parallèle. Ce chapitre résume nos différentes contributions et présente les perspectives ouvertes par ces travaux.

Conclusion générale

Les travaux présentés ici portent sur l'amélioration du processus de vérification au moyen d'automates de Büchi généralisés. Ces automates ont de nombreux avantages. Tout d'abord, ils offrent une représentation de l'automate de la propriété plus concise que ne le permettent les automates de Büchi classiques. Cette compacité est fondamentale car elle aide à combattre l'explosion combinatoire lors de la construction du produit synchronisé. Ensuite, ils sont naturellement engendrés par les algorithmes de traduction de formules de logique temporelle en automates : leur utilisation évite une opération coûteuse de dégénéralisation. Enfin, ils permettent l'expression d'hypothèses d'équité faible ou inconditionnelle à moindre coût.

Lors du processus de vérification, la prise en compte de l'équité est importante car elle n'autorise que la détection de contre-exemples réalistes. Actuellement, la majorité des *model checkers* utilise des tests de vacuité travaillant sur des automates non-généralisés. Cette thèse s'est donc attaquée à la mise en place de procédures de vérification efficaces pour ces automates.

Contribution aux tests de vacuité séquentiels Dans un premier temps, nous nous sommes focalisés sur l'étude des tests de vacuité séquentiels. Ces derniers peuvent être simplifiés en fonction de la force de l'automate de la propriété et de nombreuses optimisations existent. Afin d'avoir une vue homogène nous avons proposé la mise en place d'un canevas unifié permettant l'expression de tous ces tests.

Nous nous sommes ensuite focalisés sur les tests de vacuité supportant naturellement les automates forts et généralisés. Tous sont basés sur un calcul des composantes fortement connexes de l'automate. Nous avons alors proposé trois nouveaux tests de vacuité généralisés et comparé leurs performances relatives. Le premier est basé sur le calcul des composantes fortement connexes de Tarjan. Il se pose en alternative à la majorité des tests généralisés existants qui sont basés sur celui de Dijkstra. Les deux autres tests de vacuité proposés utilisent pour la première fois une structure d'union-find. Cette structure est particulièrement adaptée au calcul des composantes fortement connexes puisqu'elle gère des partitions. Son utilisation permet à la fois de simplifier le test de vacuité mais aussi de réduire la complexité dans le pire cas.

Lors de la mise en place de ces algorithmes nous avons également proposé une nouvelle optimisation dont peuvent bénéficier aussi bien les algorithmes de calcul de composantes fortement connexes, que les tests de vacuité. Cette optimisation compresse la pile des positions et réduit la consommation mémoire. Nous avons aussi intégré l'optimisation de Nuutila et Soisalon-Soininen [61] qui permet un gain mémoire non négligeable.

Modification de l'approche automate Dans un second temps, nous avons suggéré la modification de l'approche par automates pour le *model checking* proposée par Vardi pour qu'elle puisse tirer parti des automates multi-forces. Leur étude conduit à la mise en place d'une classification fine des composantes fortement connexes ainsi qu'à la création de plusieurs heuristiques pour les identifier. Nous avons ensuite montré que la décomposition de ces automates multi-forces en trois sous automates était possible. Trois procédures de vérification peuvent alors être lancées en parallèle. Comme chaque automate décomposé est plus petit que l'automate original, chaque produit synchronisé sera plus petit : cela permet de combattre l'explosion combinatoire.

Contribution aux tests de vacuité parallèles Les résultats positifs sur une parallélisation du test de vacuité nous ont ensuite amené à étudier les tests de vacuité parallèles. On a pu constater qu'aucun ne supportait les automates généralisés et que tous utilisaient des mécanismes lourds (synchronisations ou réparations) pour s'assurer de la validité des informations partagées entre les threads. Nous avons alors montré que ces mécanismes pouvaient être évités en utilisant une structure d'union-find *lock-free* partagée entre tous les threads. De nouveaux tests de vacuité parallèles ont alors été définis. Tous ces tests partagent des informations au travers de la structure d'union-find et limitent la redondance de travail entre les threads.

Perspectives à court terme

Cette section présente les perspectives immédiates résultant des travaux présentés dans ce manuscrit.

Variations sur les algorithmes parallèles

Les tests de vacuité parallèles proposés au chapitre 8 utilisent une structure d'union-find partagée qui permet à un thread de marquer une composante fortement connexe comme morte en une unique opération atomique. Lors de cette opération, tous les états de la composante sont ensuite parcourus pour être supprimés des structures locales (lignes 4 à 8 de la procédure `markdead`, algorithme 6 page 135). Au chapitre 5, nous avons montré que ce second parcours pouvait être évité en remplaçant ces structures par un union-find local.

La construction de tests de vacuité parallèles avec un union-find local et un union-find partagé est alors possible. Dans une telle approche, marquer une composante fortement connexe comme morte peut être fait en seulement deux étapes :

1. l'union de la racine de la composante à la partition contenant l'état artificiel *alldead* de l'union-find partagé (comme pour les algorithmes parallèles à base d'union-find partagé du chapitre 8) ;
2. l'union de la racine de la composante à la partition contenant l'état artificiel *alldead* de l'union-find local au thread.

Dans cette approche, les états morts sont stockés une fois globalement et une fois pour chaque thread les ayant explorés. Par opposition, les tests de vacuité parallèles proposés au chapitre 8 ne stockent ces états qu'une seule fois. Pour que cette nouvelle approche soit compétitive en mémoire, les threads ne doivent pas stocker localement les états morts. Une suppression retardée, à la manière de l'idée proposée en section 4.4 (page 4.4), peut alors être appliquée : les états ne sont supprimés de l'union-find local qu'à l'insertion d'un nouvel état. Une telle suppression permet de libérer la mémoire uniquement lorsque cela est nécessaire.

De la même manière que dans le chapitre 5, deux tests de vacuité peuvent être construits en fonction de l'algorithme de calcul des composantes fortement connexes sur lequel ils sont basés. L'idée générale est d'utiliser les tests reposant sur un union-find local proposés dans le chapitre 5 : à chaque opération sur l'union-find local correspond une opération sur l'union-find partagé. Nous appelons cette nouvelle famille d'algorithmes $\texttt{uf}_2$.

$\texttt{Tarjan-uf}_2$. La stratégie 13 présente un tel test de vacuité basé sur l'algorithme de calcul des composantes fortement connexes de Tarjan. Les lignes marquées d'une étoile (en rouge) mettent en évidence les différences par rapport à la stratégie 7. L'unique modification sur les structures de données manipulées concerne la pile *llstack* qui ne stocke plus les ensembles d'acceptation puisque l'union-find les stocke déjà [5]. Ainsi les lignes 12, 25, 33, et 42 varient uniquement par l'absence de marques d'acceptation dans la pile *llstack*.

La modification la plus importante concerne la méthode $\texttt{GET\_STATUS}_{Tarjan-uf_2}$: pour savoir si un état est vivant il faut tester sa présence dans *ufloc*. Ce test n'est pas suffisant car un état peut être présent dans l'union-find local mais pas encore supprimé (à cause de la suppression retardée évoquée ci-dessus). Un test supplémentaire, permettant de savoir s'il est dans même partition que l'état artificiel *alldead*, est alors nécessaire. En revanche, pour savoir si un état est mort il suffit de regarder s'il est marqué comme tel dans l'union-find partagé.

Les autres modifications ont seulement trait à la gestion de l'union-find partagé. La ligne 8 déclare cette structure qui sera initialisée par l'algorithme 7 (page 135). Lors de la détection d'une transition fermante, l'état source et l'état destination sont unis avec l'ensemble d'acceptation porté par la transition (ligne 27). Comme l'union-find partagé retourne à chaque union l'ensemble d'acceptation associé à la composante fortement connexe, il suffit de tester si toutes les marques sont présentes pour détecter un cycle acceptant (ligne 28).

5. Cette modification a déjà été proposée dans la stratégie 11 pour éviter une redondance de stockage des ensembles d'acceptation.

```
Structures supplémentaires :
  struct Step {src : Q,     succ : 2^Δ,
               acc : 2^F,   pos : int} // Refinement of Step of Algo. 1
Variables Locales supplémentaires :
  ufloc : union-find of ⟨ Q ∪ {alldead} ⟩
  llstack : pstack of ⟨ p : int⟩
Variables Partagées supplémentaires :
  uf : union-find of ⟨ Q ∪ alldead, 2^F ⟩
PUSH_{Tarjan−uf2}(acc ∈ 2^F, q ∈ Q) → int
  uf.makeset(q)
  p ← ufloc.makeset(q)
  llstack.pushtransient(p)
  dfs.push(⟨ q, succ(q), acc, p ⟩)
GET_STATUS_{Tarjan−uf2}(q ∈ Q) → Status
  if ufloc.contains(q) then
    if ufloc.sameset(q, alldead) then
      return DEAD
    else
      return LIVE
  else if uf.contains(q) ∧ uf.sameset(q, alldead) then
    return DEAD
  return UNKNOWN
UPDATE_{Tarjan−uf2}(acc ∈ 2^F, dst ∈ Q)
  p← llstack.pop(dfs.top().pos)
  llstack.pushnontransient(min(p, ufloc.liveget(d)))
  ufloc.unite(dst, dfs.top().src)
  a ← uf.unite(dst, dfs.top().src, acc)
  if a = F then
    stop ← ⊤
    report Accepting cycle detected!
POP_{Tarjan−uf2}(s ∈ Step)
  dfs.pop()
  ll ← llstack.pop(s.pos)
  if ll = s.pos then
    // Mark this SCC as Dead.
    ufloc.unite(s.src, alldead)
    uf.unite(s.src, alldead)
  else
    ufloc.unite(s.src, dfs.top().src)
    a ← uf.unite(s.src, dfs.top().src, s.acc)
    p ← llstack.pop(dfs.top().pos)
    llstack.pushnontransient(min(p, ll))
    if a = F then
      stop ← ⊤
      report Accepting cycle detected!
```

Stratégie 13: $\texttt{uf}_2$ basé sur l'algorithme de Tarjan.

```
Structures supplémentaires :
  struct Step {src : Q,  succ : 2^Δ,  acc : 2^F }          // Refinement of Step of Algo. 1

Variables Locales supplémentaires :
  ufloc : union-find of ⟨ Q ∪ {alldead} ⟩
  rstack : pstack of ⟨ p : int, acc : 2^F ⟩

Variables Partagées supplémentaires :
  uf : union-find of ⟨ Q ∪ alldead, 2^F ⟩

PUSH_{Dijkstra−uf_2}(acc ∈ 2^F, q ∈ Q) → int
  uf.makeset(q)
  ufloc.makeset(q)
  rstack.pushtransient(dfs.size())
  dfs.push(⟨ q, succ(q), acc ⟩)

GET_STATUS_{Dijkstra−uf_2}(q ∈ Q) → Status
  if ufloc.contains(q) then
    if ufloc.sameset(q, alldead) then
      return DEAD
    else
      return LIVE
  else if uf.contains(q) ∧ uf.sameset(q, alldead) then
    return DEAD
  return UNKNOWN

UPDATE_{Dijkstra−uf_2}(acc ∈ 2^F, dst ∈ Q)
  ⟨r, a⟩ ← rstack.pop(dfs.size() -1)
  a ← a ∪ acc
  while ¬ ufloc.sameset(d, dfs[r].src) do
    ufloc.unite(d, dfs[r].src)
    a ← uf.unite(d, dfs[r].src, a)
    ⟨r, la⟩ ← rstack.pop(r - 1)
    a ← a ∪ dfs[r].acc ∪ la
  rstack.pushnontransient(r, a)
  if a = F then
    stop ← ⊤
    report Accepting cycle detected!

POP_{Dijkstra−uf_2}(s ∈ Step)
  dfs.pop()
  if rstack.top(s.pos) = dfs.size() then
    rstack.pop(dfs.size())
    // Mark this SCC as Dead.
    ufloc.unite(s.src, alldead)
    uf.unite(s.src, alldead)
```

Stratégie 14: $\mathtt{uf}_2$ basé sur l'algorithme de Dijkstra.

Dijkstra-uf$_2$. La stratégie 14 présente un test de vacuité fonctionnant sur le même principe mais basé sur l'algorithme de calcul des composantes fortement connexes de Dijkstra. Les lignes marquées d'une étoile (en rouge) mettent en évidence les différences par rapport à la stratégie 8. L'unique modification concerne la gestion de l'union-find partagé. La méthode GET_STATUS$_{Dijkstra-uf_2}$ est modifiée de la même manière que pour la stratégie précédente puisque le statut d'un état est récupéré en deux temps : d'abord localement dans *ufloc*, puis globalement dans *uf*. La découverte d'un nouvel état conduit à son insertion dans l'union-find partagé (ligne 9). Lors de la détection d'une nouvelle racine, toutes les racines potentielles sont fusionnées au sein de l'union-find partagé (ligne 27). Enfin, la détection d'une racine conduit à son union avec la partition *alldead* (ligne 40).

Note : Les deux algorithmes présentés ici ont exactement le même fonctionnement que ceux proposés au chapitre 8 (pages 134 et 139). Ainsi, dans l'algorithme Tarjan-uf$_2$ toutes les informations sont publiées dans l'union-find dès qu'elles sont découvertes localement par un thread. Dans l'algorithme Dijkstra-uf$_2$, seule la découverte d'une racine « plus ancienne » (pour un thread donné) permet l'union de plusieurs états et la mise à jour de l'ensemble d'acceptation dans l'union-find partagé.

L'union-find comme support de partage

Dans ce manuscrit, nous avons proposé plusieurs algorithmes parallèles qui utilisent tous une structure d'union-find pour partager de l'information entre les différents threads. Dans cette section nous montrons comment cette structure peut être utilisée pour amplifier ce partage. Les idées proposées ici ne sont pas encore arrivées pleinement à maturité et leur compatibilité avec les autres techniques permettant de combattre l'explosion combinatoire n'a donc pas été étudiée.

Exploiter les états vivants. Les algorithmes parallèles proposés dans ce manuscrit insèrent les nouveaux états dans l'union-find partagé dès leur découverte. À un instant donné, l'ensemble des états qui ne sont pas dans la même partition que l'état artificiel *alldead* constitue l'union des états vivants de tous les threads. Cette information peut être exploitée pour diriger le parcours des threads.

Lorsqu'un thread détecte un état s qu'il n'a pas encore traité, il essaye de l'insérer dans l'union-find. Si cet état est déjà présent, cela signifie qu'un autre thread est en train de le traiter. Dans ce cas, le thread ayant tenté l'insertion peut retarder le traitement de s : il lui suffit alors de réorganiser les transitions du prédécesseur de s pour forcer s à être le dernier successeur visité. Ce thread va alors être orienté en priorité vers des parties de l'automate qui n'ont pas été explorées.

Partager les états vivants au sein des tests de vacuité parallèles a déjà été exploité pour les algorithmes basés sur un NDFS, mais conduit généralement à la mise en place de procédures de réparation ou de synchronisation [27, 28] qui peuvent limiter le passage à l'échelle. L'approche proposée ici est optimiste puisqu'elle espère que les états dont le traitement a été retardé seront traités par un autre thread.

Jouer sur les permutations. Une grande partie des algorithmes parallèles utilisent le SWARMING pour maximiser la répartition des threads lors de l'exploration. Cette technique est cependant aléatoire, et rien n'assure que deux threads ne vont pas être orientés dans la même composante fortement connexe. Si tel est le cas, les threads risquent de travailler sur les mêmes états et donc augmenter la contention sur la structure d'union-find.

Comme les composantes fortement connexes de l'automate du produit sont nécessairement synchronisées avec des composantes de l'automate de la propriété, le SWARMING peut travailler directement sur les transitions de cet automate. Ainsi, les transitions peuvent être regroupées en fonction de la composante fortement connexe vers laquelle elles vont. Le SWARMING se charge alors seulement de réorganiser ces groupes pour s'assurer que les threads sont orientés vers des composantes différentes.

Approche dynamique Dans le chapitre 8, dès qu'un thread détecte que deux états sont dans la même composante fortement connexe, il les unit dans la structure d'union-find partagée. Dans la stratégie 12 (basée sur l'algorithme de Dijkstra) le nombre d'unions effectuées par un thread est minimal : il n'effectuera l'union de deux états qu'une seule et unique fois. Lors de cette union, si les deux états sont déjà dans la même partition de l'union-find, cela signifie qu'un autre thread est en train de parcourir cette composante. Dans ce cas, un système d'attribution peut être envisagé : si deux threads se rendent compte qu'ils visitent la même composante, cette dernière est attribuée à un thread. Les threads qui ne doivent pas gérer la composante ne publieront aucune information la concernant dans l'union-find : cela permet ainsi de minimiser la contention sur cette structure.

Approche asynchrone Dans les tests de vacuité parallèles, la compatibilité avec le *Bit State Hashing* ou le *State Space Caching* est compliquée à mettre en œuvre mais est néanmoins possible. Supposons que la structure d'union-find ne soit plus une structure partagée mais une structure locale à un thread dont l'objectif est d'effectuer les unions pour tous les autres threads. Ce thread effectue alors les unions et dès qu'il détecte une partition dans laquelle toutes les marques d'acceptation sont présentes, un contre-exemple est reporté. Nous avons montré au chapitre 5 la compatibilité de l'union-find avec le *Bit State Hashing* et le *State Space Caching*. Les autres threads utilisent alors une structure dédiée dans laquelle ils vont publier les unions qui seront traitées de manière asynchrone par le thread qui en a la charge. Cette structure peut être composée d'une liste par thread. Ces listes, qui contiennent les unions à effectuer, fonctionnent alors dans un schéma producteur-consommateur et la contention y est faible.

Parallélisme et décomposition

Dans le chapitre 7, nous avons proposé une décomposition de l'automate de la propriété en trois automates représentant les comportements terminaux, faibles et forts. Dans cette section nous proposons certaines pistes pour montrer comment les tests de vacuité parallèles peuvent exploiter cette décomposition et comment celle ci peut être raffinée.

Décomposition et algorithmes parallèles. Dans le chapitre 7, les algorithmes utilisés pour tester la vacuité de chaque automate sont séquentiels mais l'utilisation d'algorithmes parallèles est possible. Pour un nombre de threads donné (nth), plusieurs stratégies sont envisageables :

1. les tests de vacuité sont séquentialisés : un ordre est fixé et l'ensemble des threads sont utilisés pour chaque test ;
2. les tests sont lancés en parallèles et chaque test utilise $nth/3$ threads ;
3. les tests sont lancés en parallèles et chaque test utilise $nth/3$ threads. Lorsqu'un algorithme termine, les autres threads peuvent aller aider les tests de vacuité restants. Pour cela, il leur suffit de récupérer un état vivant, le considérer comme l'état initial et lancer le test de vacuité adapté.

Ces trois stratégies permettent d'exploiter finement les automates multi-forces tout en tirant partie des threads disponibles pour accélérer le test de vacuité.

Amélioration de la décomposition. La décomposition proposée au chapitre chapitre 7 produit au maximum trois automates ce qui limite le passage à l'échelle. L'automate de la propriété peut néanmoins être décomposé de manière plus fine en extrayant tous les cycles acceptants des composantes fortement connexes. Il suffit ensuite de rajouter tous les chemins depuis l'état initial vers un des états de ce cycle pour construire un automate reconnaissant une partie du langage original. Une fois tous ces automates construits, leur vacuité peut être testée en parallèle en utilisant le test de vacuité le plus adapté. De la même manière que pour le chapitre 7, dès qu'un contre-exemple est trouvé tous les threads peuvent s'arrêter, sinon il faut attendre la fin du dernier test.

Perspectives à long terme

Les travaux présentés dans ce manuscrit se sont focalisés sur les automates de Büchi généralisés qui permettent une représentation concise des formules de logique temporelle linéaire et une représentation compacte de l'équité. L'équité forte peut néanmoins être représentée de manière encore plus compacte via l'utilisation d'automates de Rabin ou de Streett. Ces automates différent des TGBA, par leur interprétation des marques d'acceptation et il n'existe pas, à notre connaissance, d'algorithmes parallèles permettant de tester leur vacuité. Ces automates sont pourtant massivement utilisés dans le cadre de la théorie des jeux et la mise en place de tels algorithmes constituerait un axe de recherche intéressant.

Nous nous sommes focalisés ici sur le *model cheking* explicite mais d'autres approches existent. Les approches hybrides [23, 50], permettent une représentation compacte de certaines parties de l'espace d'état du produit. L'utilisation d'une structure d'union-find basée sur des diagrammes de décision permettrait d'envisager une approche hybride totalement nouvelle. Cette structure pourrait être ensuite parallélisée et combinée avec les travaux de Duret-Lutz et al. [23] pour permettre l'introduction d'algorithmes parallèles supportant tout LTL dans le cadre des approches hybrides ou symboliques.

Nous avons évoqué, dans ce manuscrit, la compatibilité des tests de vacuité avec les techniques de *Bit State Hashing* et de *State Space Caching*. Lorsque le système résulte de l'entrelacement de plusieurs processus, des techniques de réduction basées sur de l'ordre partiel [39] peuvent être appliquées. Ces techniques ne fonctionnent que sur des propriétés insensibles au bégaiement (i.e. dans lesquelles toute lettre peut être répétée infiniment sans changer d'état). Une décomposition des automates de la propriété en une partie sensible au bégaiement et une qui n'est pas sensible permettrait d'appliquer au moins partiellement ces techniques. De la même manière, une activation dynamique de ces techniques pour les parties de l'automate qui ne sont pas sensibles au bégaiement permettrait une amélioration des tests de vacuité. Une étude approfondie de l'application de ces techniques, aussi bien en séquentiel qu'en parallèle, constituerait une avancée significative dans la vérification efficace des propriétés LTL.

Annexe A

Détails d'implémentation et conditions d'évaluation

Sommaire

Dans les chapitres 6 et 10 nous avons évalué les performances des tests de vacuité séquentiels et parallèles. Ce chapitre détaille certains aspects liés à leur implémentation et présente les conditions dans lesquelles les expérimentations ont été réalisées.

A.1 Détails d'implémentation

Tous les tests de vacuité présentés aux chapitres 3, 5 et 9 ont été implémentés dans Spot [1] qui est est une bibliothèque dédiée au *model-checking*. Cette bibliothèque est centrée autour des TGBA et fournit les briques de bases nécessaires à la construction de *model-checkers*. Dans cette bibliothèque, les marques d'acceptation et les étiquettes portées par les transitions sont représentées au moyen de BDD [2]. Les BDD qui sont utilisés ne sont pas *threads-safe*, ce qui veut dire qu'ils ne peuvent pas être utilisés dans un contexte parallèle.

Afin de palier à cela, nous avons commencé par redéfinir [3] un nouveau type de TGBA qui utilise des vecteurs de bits plutôt que des BDD. Ces vecteurs sont particulièrement utiles puisqu'ils peuvent être vus comme des entiers et manipulés concurremment. Nous avons alors effectué une migration de Spot pour rendre l'outil compatible avec la norme C++11. Cette norme introduit les threads comme standard du C++ et semble particulièrement adaptée pour développer un outil multi-plateformes.

Afin de pouvoir vérifier des propriétés sur des modèles, nous avons aussi du réécrire l'interface entre Spot et Divine2.4 (patchée par LTSmin). Cet outil permet de générer une bibliothèque dynamique à partir d'un modèle exprimé en DVE. Cette bibliothèque peut être manipulée au travers d'une dizaine de méthodes prédéfinies et représente les comportements du modèle sous

1. http://spot.lip6.fr/wiki/
2. http://buddy.sourceforge.net/
3. L'outil peut être téléchargé http://pagesperso-systeme.lip6.fr/Etienne.Renault/phd/sumup.html

la forme d'un tableau d'entier. Le choix d'une interface avec un outil supportant DVE a été fait pour pouvoir bénéficier des nombreux modèles présents dans le jeu de test BEEM. De plus, cela permet de comparer aisément les outils et leurs résultats.

Pour évaluer les tests de vacuité parallèle nous avons du mettre en place la structure d'union-find partagée. Pour cela, nous avons récupéré la table de hachage *lock-free* de LTSmin [4] qui offre de bonnes performances [55]. Les relations de parentés entre les différentes partitions de l'automate ont ensuite été réalisées via des listes chaînées qui sont facilement implémentables au moyen d'opérations atomiques. Dans cette structure les chemins sont compressés à chaque opération `find`. Notons aussi que chaque élément de la liste contient, en plus du lien de parenté, un entier qui stocke les marques d'acceptations. Modifier cet ensemble peut être fait en trois étapes : (1) il faut récupérer le représentant de la partition, (2) il faut modifier l'ensemble d'acceptation, et (3) il faut vérifier que le représentant n'a pas changé. S'il a changé, il suffit de répéter les opérations précédentes [5].

Nous avons évalué que la mise en place de l'ensemble des travaux de cette thèse à nécessité 14 000 lignes de code sur les 125 000 lignes que compte Spot.

A.2 Conditions d'évaluation

Afin de pouvoir évaluer les différents tests de vacuité de manière juste nous avons utilisés la même configuration matérielle pour toutes les expérimentations. Pour cela nous avons utilisé une machine composée de quatre Intel(R) Xeon(R) CPUX7460@ 2.66GHz disposant d'une mémoire de 132 Go. De plus, chaque test de vacuité a été borné à une heure [6] et nous n'avons pas détecté de dépassement mémoire. Tous les outils ont été compilés avec GCC 4.8.2 et nous avons particulièrement pris soin d'activer toutes les optimisations possibles lors de la compilation des différents outils.

Nous présentons ici les différents paramètres que nous avons pris pour les différents outils :

- tous les algorithmes (à l'exception de `cndfs` et `owcty`) traduisent la formule LTL en un automate en utilisant la traduction proposée par [19]. Après cette traduction sont appelées des opération de post-réductions qui visent à réduire au maximum la taille de l'automate. Lors de la décomposition, cette opération est appliquée sur chaque automate décomposé ;
- l'algorithme `owcty` est lancé en deux temps. Tout d'abord l'automate de la formule est généré dans un fichier qui contient aussi le modèle à l'aide de la commande *divine combine model -f file.ltl* (où file.ltl représente le fichier contenant la formule à vérifier). Le fichier model.prop1.dve est alors généré.

  Ensuite pour vérifier la propriété en utilisant N thread la commande suivant est lancée : *divine owcty -w N -r model.prop1.dve.*
- l'algorithme `cndfs` est lui aussi lancé en deux temps. La première phase est identique à celle mentionnée ci-dessus et produit un fichier contenant l'automate de la formule à vérifier et le modèle en utilisant l'outil Divine.

4. Cette table est inspirée d'une implémentation C++ de la table de hachage proposée par Cliff Click pour Java. Plus de détails : https://code.google.com/p/nbds/
5. Nous choisissons comme représentant l'élément de la liste des représentants qui a la plus petite adresse mémoire, ce qui nous donne un ordre total.
6. À l'aide de l'outil unix *timeout*.

Ensuite, l'algorithme est lancé pour N threads au moyen de la commande *dve2lts-mc –ltl-semantics=spin -p=random –threads=N –strategy=cndfs –state=table -v model.prop1.dve*

Annexe B

Preuve des tests de vacuité parallèles généralisés

Cette annexe présente la preuve de justesse du test de vacuité parallèle basé sur l'algorithme de calcul des composantes fortement connexes de Dijkstra (cf 9). Un raisonnement similaire peut être appliqué pour prouver la justesse du test de vacuité basé sur l'algorithme de calcul des composantes fortement connexes de Tarjan.

Cette annexe détaille la preuve de justesse du test de vacuité présenté en stratégie 12 (page 139). Pour faciliter la lecture de cette preuve, l'algorithme B.1 reprend cet algorithme et le simplifie. Toutes les modifications apportées sont de forme mineures et n'affectent pas le comportement du test de vacuité. Les principaux changements sont :

1. la fusion du DFS générique, de la procédure principale, de la méthode `markdead`, et de la spécialisation de l'algorithme parallèle. La principale conséquence est que le cœur de l'algorithme (lignes 19 à 34) utilisent directement les spécialisations ;
2. la suppression du paramètre permettant de choisir la politique d'ordonnancement des successeurs. Comme les algorithmes parallèles utilisent tous la politique RANDOM, nous intégrons directement cette modification à la ligne 24 ;
3. la pile *rstack* n'intègre plus l'optimisation proposée à la section 4.4 (ligne 14). Ainsi les opérations effectuées sur cette pile ne sont pas masquées derrière une interface générique. Les lignes 50 et 56 sont légèrement modifiées et l'opération `pop` requise par l'optimisation de la pile compressée est simplement remplacée par `top` : cela n'est pas gênant car on remplace, un dépilement et un empilement successifs, par une simple consultation de la valeur au sommet de la pile.

Avant toute chose, nous rappelons ici le rôle des différentes variables et structures manipulées par cet algorithme :

- *uf* : l'union-find *lock-free* partagé par tous les threads. Il maintient pour chaque partition un ensemble de marques d'acceptations ;
- *stop* : la variable booléenne partagée permettant de forcer l'arrêt de tous les threads ;
- *dfs* : la pile locale représentant la pile DFS ;
- *rstack* : la pile locale symbolisant la pile des racines. Elle référence les positions dans la pile *dfs* des racines potentielles ;

```
Variables Partagées :
  𝒜 : TGBA such that 𝒜 = ⟨ Q, q0, AP, ℱ, Δ ⟩
  stop : boolean
  uf : union-find of ⟨ Q ∪ alldead, 2^ℱ ⟩
Structures Globales :
  struct Transition {src : Q, acc : 2^ℱ, dst : Q}
  struct Step       {src : Q,   succ : 2^Δ,
                     acc : 2^ℱ,  pos : int}
  enum Status       {LIVE, DEAD, UNKNOWN}
Variables Locales :
  dfs : stack of ⟨ Step ⟩
  live : stack of ⟨ Q ⟩
  livenum : map of Q ↦ ⟨p : int⟩
  rstack : stack of ⟨ p : int, acc : 2^ℱ ⟩

parallel_main()
  stop ← ⊥
  makeset(alldead)
  DijkstraParEC() || ... || DijkstraParEC()

DijkstraParEC()
  PUSH_DijkstraPar(∅, q0)
  while ¬ dfs.empty() ∧ ¬ stop do
    Step step ← dfs.top()
    if step.succ ≠ ∅ then
      Transition t ← randomly pick one from step.succ
      switch GET_STATUS_DijkstraPar(t.dst) do
        case DEAD
          skip
        case LIVE
          UPDATE_DijkstraPar(t.acc, t.dst)
        case UNKNOWN
          PUSH_DijkstraPar(t.acc, t.dst)
    else
      POP_DijkstraPar(step)
  stop ← ⊤

PUSH_DijkstraPar(acc ∈ 2^ℱ, q ∈ Q) → int
  uf.makeset(q);
  p ← livenum.size();
  livenum.insert(⟨ q, p ⟩);
  rstack.push(⟨dfs.size(), ∅⟩);
  dfs.push(⟨ q, succ(q), acc, p ⟩);

GET_STATUS_DijkstraPar(q ∈ Q) → Status
  if livenum.contains(q) then
    return LIVE
  else if uf.contains(q) ∧
        uf.sameset(q, alldead) then
    return DEAD
  return UNKNOWN

UPDATE_DijkstraPar(acc ∈ 2^ℱ, dst ∈ Q)
  dpos ← livenum.get(dst);
  ⟨r, a⟩ ← rstack.top();
  a ← a ∪ acc
  while dpos < dfs[r].pos do
    ⟨r, la⟩ ← rstack.pop();
    a ← a ∪ dfs[r].acc ∪ la;
    a ← unite(dst, dfs[r].src, a)
  rstack.top().acc ← a;
  if a = ℱ then
    stop ← ⊤;
    report Accepting cycle detected !

POP_DijkstraPar(s ∈ Step)
  dfs.pop();
  if rstack.top() = dfs.size() then
    rstack.pop() ;
    markdead(s);
  else
    live.push(s.src);

markdead(s : Step)
  uf.unite(s.src, alldead);
  livenum.remove(s.src);
  while livenum.size() > s.pos do
    q ← live.pop();
    livenum.remove(q);
```

FIGURE B.1 – Algorithme parallèle basé sur Dijkstra (expansé).

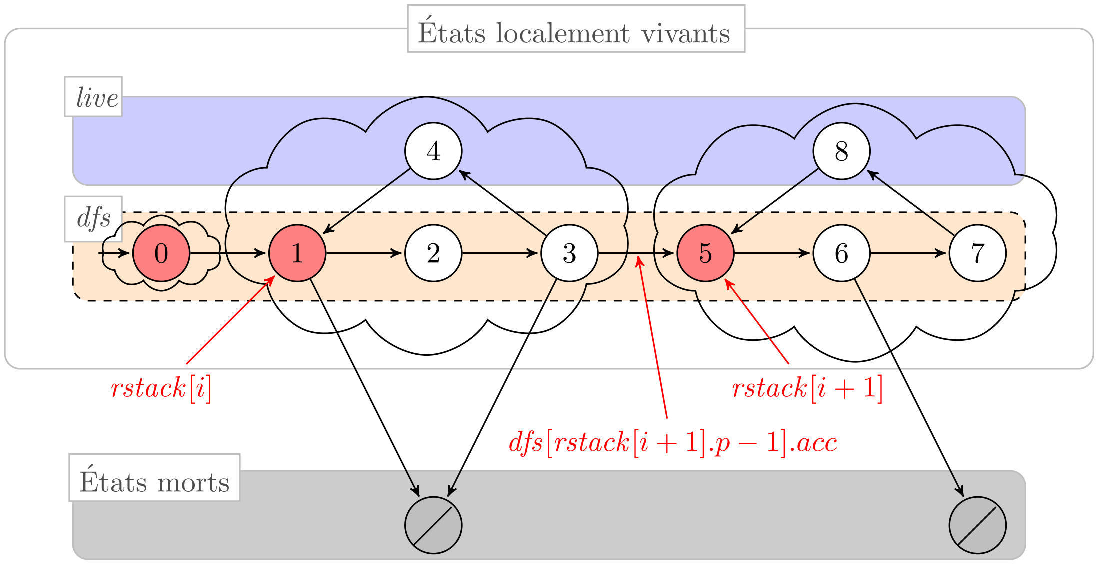


Figure B.2 – Illustration des notations utilisées dans cette annexe.

livenum : le tableau associatif locale associant chaque état vivant à un LIVE *number* ;

live : la pile locale stockant les états vivants qui ne sont plus sur la pile *dfs*.

Nous souhaitons prouver que : « *ce test de vacuité ne détecte pas de contre-exemple et explore l'intégralité de l'espace d'état si le langage de l'automate $\mathcal{A}$ est vide, sinon il détecte la présence d'un cycle acceptant* ». Exprimé différement cela revient à prouver les deux théorèmes suivants :

Théorème 1. Pour un TGBA $\mathcal{A}$ donné, l'algorithme détecte un contre-exemple si et seulement si $\mathscr{L}(\mathcal{A}) \neq \emptyset$.

Théorème 2. Pour un TGBA $\mathcal{A}$ donné, l'algorithme termine toujours.

Pour prouver ces deux théorèmes, nous introduisons les définitions et notions suivantes :

- un état est dit *localement vivant* pour un thread s'il est présent dans le tableau associatif *livenum* de ce thread ;
- un état est dit *mort* s'il est présent dans la structure d'union-find partagé (*uf*) et qu'il est dans la même partition que l'état artificiel *alldead* ;
- pour un thread nous notons n la taille courante de sa pile *rstack*, i.e. $n = rstack.\mathtt{size()}$;
- $\forall\, i, 0 \leq i < n$, nous notons S_i l'ensemble des états appartenant à la composante fortement connexe partielle représentée par *rstack*$[i]$, i.e. :

$$\begin{cases} S_i & = \{q \in Q \mid dfs[rstack[i].p].pos \ \leq \ livenum.\mathtt{get}(q) \ \leq \ dfs[rstack[i+1].p].pos\} \ \textbf{si}\ i \neq n-1 \\ S_{n-1} & = \{q \in Q \mid dfs[rstack[n-1].p].pos \ \leq \ livenum.\mathtt{get}(q)\ \textbf{sinon}. \end{cases}$$

Ces notions sont représentées schématiquement sur la figure B.2. Les états sont étiquetés par leur LIVE *number* et sont donc présents dans *livenum*. Les états morts sont barrés et grisés puisqu'ils n'ont pas de LIVE *number* associé. Les états rouges de la pile *dfs* constituent les racines potentielles. Les états 4 et 8 sont vivants mais plus sur la pile *dfs* : ils sont donc dans *live*. Les nuages représentent les ensembles S_i, et on pour un S_i donné tous les états sont dans la même partition de l'union-find. Enfin, les annotations rouges représentent les variables manipulées par les définitions ci-dessus.

Pour prouver la justesse l'algorithme B.1, supposons tout d'abord que chaque ligne s'exécute de manière atomique. Cette supposition est réaliste puisque : (1) la structure d'union-find est *lock-free*, (2) l'unique variable partagée en lecture/écriture est *stop* qui est un entier et peut donc être modifié de manière *lock-free* en utilisant des instructions *compare-and-swap*, et (3) toutes les autres variables sont locales et ne peuvent donc pas être modifiées par les autres threads. En utilisant cette hypothèse, nous montrons que les invariants suivants sont préservés par l'algorithme :

Lemme 1. *La variable rstack désigne un sous-ensemble de la pile dfs et rstack contient des valeurs croissantes.*

Démonstration. Le lemme 1 est vérifié par définition puisque la pile *rstack* contient des positions de la pile *dfs* : il s'agit donc d'un sous-ensemble de *dfs*. Les valeurs qu'elle contient sont croissantes car chaque ajout ligne 39 est fait en concordance avec la taille de la pile *dfs* qui est augmentée ligne 40. De plus, la taille de *rstack* n'est diminuée ligne 63 qu'après avoir diminué la taille de *dfs*. Enfin, l'opération de la ligne 53 ne fait que supprimer l'élément au sommet : la propriété de croissance est maintenue puisque la pile *rstack* a une taille inférieure ou égale à la pile *dfs*. □

Lemme 2. *Il existe une transition portant la marque d'acceptation $dfs[rstack[i+1].p].acc$ entre les composantes partielles d'indice i et $i+1$. Autrement dit :*

$$\forall\, 0 < i < n-1, \exists\, \ell \in 2^{AP},\ \text{tel que } (dfs[pos-1].src,\ \ell,\ dfs[pos].acc,\ dfs[pos].src) \in \Delta,\ \text{où } pos = rstack[i+1].p$$

Démonstration. Lors de la première opération $\texttt{PUSH}_{DijkstraPar}$ à la ligne 20, l'état initial est ajouté dans les variables locales *dfs* et *rstack*. On a donc $n = 1$, et le lemme est vérifié de manière triviale. Les autres opérations $\texttt{PUSH}_{DijkstraPar}$ à la ligne 31 modifient les variables *rstack* et *dfs* aux lignes 39 et 40. Comme ce sont deux opérations locales, il suffit de vérifier que le lemme est vérifié avant et après. À la ligne 39 une nouvelle position est insérée dans *rstack*. Cette position référence l'état qui est inséré dans *dfs* à la ligne 40. Par définition (et comme n vient d'être augmenté), $dfs[rstack[n-1].p].acc$ et $dfs[rstack[n-1].p].src$ correspondent aux paramètre de la méthode $\texttt{PUSH}_{DijkstraPar}$, tandis que l'attribution des positions (ligne 39) assure que pour $i = n-2$ alors $dfs[rstack[i+1].p-1].src$ est l'ancien élément au sommet de la pile *dfs*. Nous avons donc montré que tout appel à $\texttt{PUSH}_{DijkstraPar}$ met correctement à jour les variables *rstack* et *dfs*. Il ne reste plus qu'à montrer qu'une telle transition existe dans $\mathcal{A}$. Comme la méthode $\texttt{PUSH}_{DijkstraPar}$ n'est déclenchée que lors de la détection d'une transition dont la destination n'est pas dans *livenum*, on a l'assurance que cette transition existe. Le lemme 2 est vérifié avant et après les appels à $\texttt{PUSH}_{DijkstraPar}$.

Lors d'un appel à $\texttt{UPDATE}_{DijkstraPar}$, la ligne 53 dépile des éléments de *rstack*. Le seul impact de cette ligne est la diminution de n, mais le lemme reste vérifié puisqu'il l'était pour un n plus grand. De la même manière, lors d'une opération $\texttt{POP}_{DijkstraPar}$ ce lemme reste vérifié. En effet, le sommet de *rstack* stocke au maximum une référence vers le sommet de la pile *dfs* : on doit donc s'assurer que la position référencée est valide (i.e., pointe toujours vers un élément de *dfs*). D'après le lemme 1 on sait que les positions stockées sont croissantes, et d'après la ligne 39 que cette position ne peu excéder la taille de la pile *dfs*. Dans le pire cas, les piles *dfs* et *rstack* sont dépilées lignes 61 et 63 lors d'un même appel à $\texttt{POP}_{DijkstraPar}$. La position stockée par *rstack* ne peut excéder la taille de *dfs* : le lemme 2 est donc vérifié. □

Lemme 3. *$\forall\, 0 \leq i < n$, le sous-graphe induit par S_i est une composante fortement connexe partielle.*

Démonstration. Nous nous focalisons ici sur les lignes modifiant la variable *rstack*. À la ligne 20 la première opération $\texttt{PUSH}_{DijkstraPar}$ ajoute l'état initial dans les variables locales *dfs* et *rstack* (lignes 39 et 40) : l'unique élément de *rstack* représente une composante fortement connexe partielle composée de l'état initial. En conséquence, le lemme 3 est trivialement vérifié. De la même manière, ce lemme est vérifié pour chaque opération $\texttt{PUSH}_{DijkstraPar}$ de la ligne 31 puisqu'il s'agit de la création d'une composante partielle composé d'un unique état.

La variable *rstack* est aussi modifiée à la ligne 53, lors d'une opération $\texttt{UPDATE}_{DijkstraPar}$. Cette opération n'est déclenchée qu'à la détection d'une transition fermante (vers un état localement vivant). Lors de cette opération, on a l'assurance que l'élément qui va être dépilé de *rstack* appartient à une composante fortement connexe partielle dont il n'est pas la racine. En effet, la méthode $\texttt{UPDATE}_{DijkstraPar}$ est déclenchée lors de la détection d'une transition fermante : cela implique que le LIVE *number* de la destination de la transition est plus petit que celui de la source. Ces deux états sont donc dans la même composante fortement connexe partielle et la position associée à l'état source peut être supprimée de *rstack*(ligne 53) sans invalider les S_i. □

Lemme 4. *Si la partition à laquelle appartient un état est associée à $acc \neq \emptyset$ dans l'union-find partagé, alors la composante fortement connexe contenant cet état possède un cycle visitant acc. (Note : si un état est uni avec alldead, il est toujours associé à $\emptyset$.)*

Démonstration. Par définition, le premier appel à `makeset` pour un état l'associe à un ensemble d'acceptation vide (ligne 36). De la même manière, tous les états unis avec *alldead* (ligne 69) sont associés avec un ensemble d'acceptation vide. Nous nous focalisons donc sur l'unique endroit où la marque d'acceptation d'une partition peut être modifiée, c'est à dire ligne 55 lors d'une union. Cette union n'est réalisée qu'à la détection d'une transition fermante, i.e. lorsque deux états font parti de la même composante fortement connexe (d'après le lemme 3). Lors de cette union l'ensemble d'acceptation calculé à la ligne 54 est alors passé en paramètre. Deux cas sont alors distingués :

1. si cet ensemble est vide et que l'ensemble retourné par la méthode `unite` l'est aussi : le lemme 4 est vérifié trivialement.
2. sinon la variable a (ligne 54) représente un sous ensemble des marques d'acceptations de la composante fortement connexe. Lors de l'opération `unite`, il va venir augmenter celui présent dans l'union-find (de la même partition). Comme cet ensemble ne peut avoir été modifié que par la ligne 55 d'un autre thread, on a l'assurance qu'il existe un cycle de $\mathcal{A}$ qui visite ces marques. L'accumulation de ces marques avec celles calculées localement assure qu'il existe un cycle visitant cet union de marques : le lemme 4 est vérifié. L'accumulation de la ligne 56 est alors valide.

□

Proposition 1. *Si l'algorithme signale un contre-exemple, celui-ci existe.*

Démonstration. La détection d'un contre-exemple ne peut se faire qu'aux lignes 57 à 60. Cette détection est conditionnée par la valeur du champ *acc* au sommet de la pile *rstack*(qui est équivalent à a, ligne 56). D'après les lemmes 1, 2, 3 et 4 on sait qu'il existe un cycle de $\mathcal{A}$ visitant toutes ces marques. □

Lemme 5. *Le premier thread qui marque un état comme mort a vu toute la composante fortement connexe à laquelle cet état appartient.*

Démonstration. Un état ne peut être déclaré comme mort que lors d'un appel à la méthode `markdead` (lignes 67 à 72). Un thread ne peut appeler cette méthode que s'il détecte que le sommet de la pile *rstack* est égal à la taille de la pile *dfs*. Au moment de l'union avec l'état artificiel *alldead* (ligne 68), le thread a vu tous les états de S_i (qui est une composante fortement connexe partielle d'après le lemme 3) et tous leurs descendants non marqués comme morts (d'après les lignes 26 et 27 qui ignorent les états morts). On distingue alors deux cas :

1. soit c'est le premier thread qui marque un état de cette composante comme mort et alors S_i contient tous les états de la composante. En effet, si un état de la composante n'a pas été vu, cela signifie qu'il a été marqué comme mort par un autre thread (d'après la ligne 47) et il y a contradiction ;
2. sinon, ce n'est pas le premier thread.

Dans les deux cas, le lemme 5 est vérifié. □

Lemme 6. *L'ensemble des états morts est une union de composantes fortement connexes maximales, i.e. l'ensemble maximal d'état pouvant former une composante fortement connexe partielle.*

Démonstration. Après un appel à $\texttt{UPDATE}_{DijkstraPar}$, le sommet de la pile *rstack* contient la position dans *dfs* de l'état (de la composante fortement connexe) avec le plus petit LIVE *number*, i.e. la racine potentielle. Pendant cet appel à $\texttt{UPDATE}_{DijkstraPar}$, tous les états de *dfs* qui ont un LIVE *number* supérieur à celui de la racine potentielle sont unis dans l'union-find (lignes 52 à 55). Si un thread visite toutes les transitions d'une composante fortement connexe on a l'assurance que tous les états de cette composante seront unis dans l'union-find. D'après le lemme 5, on sait que le premier thread qui marque un état d'une composante fortement connexe comme mort a vu toute la composante. Lorsqu'il marque un état de cette composante comme morte, c'est l'ensemble de la composante qui est marquée comme morte puisque tous ses états sont dans la même partition de *uf*. La première union avec l'état artificiel *alldead* unit toujours toute la composante. □

Lemme 7. *Si un état est mort, il ne peut participer à aucun chemin acceptant.*

Démonstration. D'après le lemme 5, le premier thread qui marque une composante comme morte l'a entièrement visitée. Lors de cette visite il n'a pas trouvé de cycle acceptant sinon il l'aurait reporté (d'après les lemmes 1 à 4). Le lemme 7 est donc vérifié trivialement. □

Proposition 2. *Si tous les états de $\mathcal{A}$ sont morts alors il n'existe pas de contre-exemple.*

Démonstration. D'après le lemme 7, les états morts ne peuvent faire partie d'un cycle acceptant. Si tous les états de $\mathcal{A}$ sont morts cela signifie qu'aucun cycle acceptant n'a été détecté et qu'il n'existe pas de contre-exemple. □

Démonstration théorème 1. Grâce aux propositions ci-dessus on sait que : le théorème 1 est vérifié car les proposition 1 et 2 sont vérifiées. □

Démonstration théorème 2. Tous les threads effectuent un parcours DFS de l'automate qui possède un nombre d'états finis. Tous les états explorés sont systématiquement insérés dans l'union-find (ligne 36) et dans *livenum* (ligne 38). Ces états ne sont jamais supprimés de l'union-find, mais sont marqués morts lorsqu'ils sont supprimés de *livenum* aux ligne 71 et 72. Comme les états morts sont ignorés par l'algorithme et que les états localement vivant ne sont insérés qu'une unique fois (méthode $\texttt{UPDATE}_{DijkstraPar}$), les états de $\mathcal{A}$ ne sont visités qu'une unique fois par un même thread. Comme l'automate possède un nombre d'états fini : l'algorithme termine. De plus, dès qu'un contre-exemple est détecté ou qu'un thread termine (lignes 34 et 58), la variable *stop* est positionnée à $\top$ et les autres threads s'arrêteront à la prochaine itération (ligne 21). □

Comme les théorèmes 1 et 2 sont vérifiés, il vient que : « *ce test de vacuité ne détecte pas de contre-exemple et explore l'intégralité de l'espace d'état si le langage de l'automate $\mathcal{A}$ est vide, sinon il détecte la présence d'un cycle acceptant* ». De plus l'algorithme se termine toujours. L'algorihtme est donc correct.

La preuve présentée ci dessus a été faite sur le test de vacuité parallèle basé sur le calcul des composantes fortement connexes de Dijkstra. Un raisonnement similaire peut être effectué sur le test de vacuité basé sur le calcul des composantes fortement connexes de Tarjan puisque les deux algorithmes ont des fonctionnements très proches. L'unique modification concerne la définition des S_i qui ne peut plus se baser sur la pile *rstack* mais qui doit se baser sur la pile des lowlinks.

Bibliographie

[1] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440 of *LNCS*, pp. 61–76. Springer, April 2005.

[2] T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In *SPIN'13*, vol. 7976 of *LNCS*, pp. 81–98. Springer, July 2013.

[3] J. Barnat, L. Brim, and J. Stríbrná. Distributed LTL model-checking in spin. In *SPIN*, vol. 2057 of *LNCS*, pp. 200–216. Springer, May 2001.

[4] J. Barnat, L. Brim, and I. Černá. Property driven distribution of Nested DFS. In *Proc. Workshop on Verification and Computational Logic*, number DSSE-TR-2002-5 in DSSE Technical Report, pp. 1–10. University of Southampton, UK, 2002.

[5] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *ASE'03*, pp. 106–115. IEEE Computer Society, 2003.

[6] J. Barnat, L. Brim, and J. Chaloupka. From distributed memory cycle detection to parallel LTL model checking. In *FMICS'04*, vol. 133 of *ENTCS*, pp. 21–39, 2005.

[7] J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core LTL model-checking. In *SPIN'07*, pp. 187–203, 2007. Springer.

[8] J. Barnat, L. Brim, and P. Ročkai. A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties. In *ICFEM'09*, vol. 5885 of *LNCS*, pp. 407–425, 2009. Springer.

[9] F. Blahoudek, A. Duret-Lutz, M. Křetínský, and J. Strejček. Is there a best Büchi automaton for explicit model checking ? In *SPIN'14*, pp. ?– ? ACM, July 2014. To appear.

[10] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *CAV'99*, vol. 1633 of *LNCS*, pp. 222–235. Springer, 1999.

[11] L. Brim, I. Černá, P. Krcal, and R. . Pelánek. Distributed LTL model checking based on negative cycle detection. In *FSTTCS'01*, pp. 96–107, 2001.

[12] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In *FMCAD'04*, vol. 3312 of *LNCS*, pp. 352–366. Springer, November 2004.

[13] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. of the International Congress on Logic, Methodology, and Philosophy of Science, Berkley, 1960*, pp. 1–11. Standford University Press, 1962.

[14] O. Carton and M. Michel. Unambiguous büchi automata. *Theoretical Computer Science*, 297(1-3) :37–81, 2003.

[15] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *MFCS'03*, vol. 2747 of *LNCS*, pp. 318–327, Aug. 2003. Springer.

[16] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In *SPIN'03*, vol. 2648 of *LNCS*, pp. 49–73. Springer, May 2003.

[17] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6) :521–549, 1996.

[18] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In *CAV'90*, vol. 531 of *LNCS*, pp. 233–242. Springer, 1991.

[19] J.-M. Couvreur. On-the-fly verification of temporal logic. In *FM'99*, vol. 1708 of *LNCS*, pp. 253–271, Sept. 1999. Springer.

[20] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software*, vol. 3639 of *LNCS*, pp. 143–158. Springer, Aug. 2005.

[21] E. W. Dijkstra. EWD 376 : Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF, May 1973.

[22] A. Duret-Lutz. *Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents.* PhD thesis, Université Pierre et Marie Curie (Paris 6), 2007.

[23] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Self-loop aggregation product — a new hybrid approach to on-the-fly LTL model checking. In *ATVA'11*, vol. 6996 of *LNCS*, pp. 336–350, Oct. 2011. Springer.

[24] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *International Conference on Software Engineering*, pp. 3–12. IEEE Computer Society, 2007.

[25] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN'01*, vol. 2057 of *LNCS*, pp. 57–79. Springer, 2001.

[26] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2–3) :247–267, 2004.

[27] S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In *ATVA'11*, vol. 6996 of *LNCS*, pp. 381–396. Springer, 2011.

[28] S. Evangelista, A. Laarman, L. Petrucci, and J. van dẽ Pol. Improved multi-core nested depth-first search. In *ATVA'12*, vol. 7561 of *LNCS*, pp. 269–283. Springer, 2012.

[29] D. Faragò and P. H. Schmitt. Improving non-progress cycle checks. In *SPIN'09*, vol. 5578 of *LNCS*, pp. 50–67. Springer, June 2009.

[30] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4) :107–114, 2000.

[31] A. Gaiser and S. Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. In *MEMICS'09*, vol. 13 of *OASICS*. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany, Nov. 2009.

[32] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commununication of the ACM*, 7(5) :301–303, may 1964.

[33] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *SPIN'04*, vol. 2989 of *LNCS*, pp. 92–108, Apr. 2004.

[34] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference*, Lecture Notes in Computer Science, pp. 205–219. Springer, April 2004.

[35] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with tarjan's algorithm. *Theoretical Computer Science*, 345(1) :60–82, 2005.

[36] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV'95*, vol. 38 of *IFIP Conference Proceedings*, pp. 3–18, June 1996. Chapman & Hall.

[37] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Thirteenth International Symposium on Protocol Specification, Testing and Verification (PSTV'93)*, vol. C-16 of *IFIP Transactions*, pp. 109–124. North-Holland, May 1993.

[38] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2) :305–326, 1994.

[39] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *CAV'92*, vol. 663 of *LNCS*, pp. 178–191. Springer, 1992.

[40] A. Hamez. *Génération efficace de grands espaces d'états*. PhD thesis, PhD thesis, Université Pierre & Marie Curie Paris 6, Paris, France, December 2009.

[41] H. Hansen and J. Geldenhuys. Cheap and small counterexamples. In *SEFM'08*, pp. 53–62. IEEE Computer Society, Nov. 2008.

[42] G. J. Holzmann. Tracing protocols. *AT&T technical journal*, 64(10) :2413–2433, 1985.

[43] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *PSTV'87*, pp. 339–344. North-Holland, May 1987.

[44] G. J. Holzmann. *Design and Validation of computer protocols*, vol. 07632 of *Prentice Hall Software Series*. Brian W. Kernighan, 1991.

[45] G. J. Holzmann. State compression in SPIN : Recursive indexing and compression training runs. In *Proc. of the 3rd Spin Workshop (SPIN '97)*, April 1997. URL `citeseer.nj.nec.com/holzmann97state.html`.

[46] G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transaction on Software Engineering*, 33(10) :659–674, 2007.

[47] G. J. Holzmann and D. Peled. An improvement in formal verification. In *FORTE'94*, vol. 6 of *IFIP Conference Proceedings*, pp. 109–124, 1994. Chapman & Hall.

[48] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of the 2nd Spin Workshop*, vol. 32 of *DIMACS : Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, may 1996.

[49] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. *IEEE Transaction on Software Engineering*, 37(6) :845–857, 2011.

[50] K. Klai and D. Poitrenaud. MC-SOG : An LTL model checker based on symbolic observation graphs. In *ICATPN'08*, vol. 5062 of *LNCS*, pp. 288—306, June 2008. Springer.

[51] P. Krcál. Distributed explicit bounded LTL model checking. In *Electronic Notes in Theoretical Computer Science*, vol. 89, pp. 33–50. Elsevier, 2003.

[52] A. Laarman and J. van de Pol. Variations on multi-core nested depth-first search. In *PDMC*, pp. 13–28, 2011.

[53] A. Laarman, R. Langerak, J. van de Pol, M. Weber and A. Wijs. Multi-core nested depth-first search. In *ATVA'11*, vol. 6996 of *LNCS*, pp. 321–335, October 2011. Springer.

[54] A. W. Laarman. *Scalable Multi-Core Model Checking*. PhD thesis, Enschede, The Netherlands, 2014. URL fmt.cs.utwente.nl/tools/ltsmin/laarman_thesis/.

[55] A. W. Laarman, J. C. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proc. of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*, pp. 247–256, October 2010. IEEE Computer Society.

[56] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2) :125 –143, march 1977.

[57] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *SPIN'06*, vol. 1680 of *LNCS*, pp. 22–39. Springer, 1999. ISBN 978-3-540-66499-4.

[58] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN'08*, vol. 2057 of *LNCS*, pp. 80–102. Springer, May 2001.

[59] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL'85*, pp. 97–107. ACM, 1985.

[60] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC'90*, pp. 377–410, 1990. ACM.

[61] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1) :9–14, January 1994.

[62] M. M. A. Patwary, J. R. S. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *SEA'10*, vol. 6049 of *LNCS*, pp. 411–423. Springer, 2010.

[63] D. J. Pearce. An improved algorithm for finding the strongly connected components of a directed graph. Technical report, Victoria University, Wellington, NZ, 2005.

[64] R. Pelánek. Typical structural properties of state spaces. In *SPIN'04*, vol. 2989 of *LNCS*, pp. 5–22. Springer, April 2004.

[65] R. Pelánek. Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Faculty of Informatics, Masaryk University Brno, October 2006.

[66] R. Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10 :443–454, 2008.

[67] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *FMICS'05*, pp. 98–105. ACM Press, 2005.

[68] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1) :39–64, 1996.

[69] A. Pnueli. The temporal logic of programs. In *FOCS'77*, IEEE, pp. 46 – 57. IEEE Computer Society, october 1977.

[70] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16 (3) :319–348, 1984.

[71] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20 : 229–234, 1985.

[72] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In *LPAR'13*, vol. 8312 of *LNCS*, pp. 668–682. Springer, Dec. 2013.

[73] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Strength-based decomposition of the property Büchi automaton for faster model checking. In *TACAS'13*, vol. 7795 of *LNCS*, pp. 580–593. Springer, Mar. 2013.

[74] X. Renault. *Mise en œuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systèmes embarqués temps-réel répartis.* PhD thesis, UPMC, 2009.

[75] R. Saad. *Parallel Model Checking for Multiprocessor Architecture.* INSA, 2011. URL http://books.google.fr/books?id=zkzvngEACAAJ.

[76] K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Proc. of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, vol. 2250 of *Lecture Notes in Artificial Intelligence*, pp. 39–54, 2001. Springer.

[77] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS'05*, vol. 3440 of *LNCS*, Apr. 2005. Springer.

[78] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1) :67–72, 1981.

[79] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. *Electronic Notes in Theoretical Computer Science*, 89(1) :51 – 67, 2003.

[80] U. Stern and D. Dill. Parallelizing the murphy verifier. *Formal Methods in System Design*, 18(2) :117–129, 2001.

[81] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, Berichte aus der Informatik, pp. 81–90. Verlag, 1996.

[82] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[83] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2) :215–225, Apr. 1975.

[84] H. Tauriainen. Nested emptiness search for generalized Büchi automata. In *ACSD'04*, pp. 165–174. IEEE Computer Society, June 2004.

[85] H. Tauriainen. A note on the worst-case memory requirements of generalized nested depth-first search. Research Report A96, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Sept. 2005.

[86] A. Valmari. The state explosion problem. In *Lectures on Petri Nets 1 : Basic Models*, vol. 1491 of *LNCS*, pp. 429–528. Springer, 1998.

[87] M. Y. Vardi. An automata-theoretic approach to automatic program verification. In *LICS'86*, pp. 332–344. IEEE Computer Society Press, 1986.

Résumé

L'approche automate pour le *model checking* de propriétés temporelles à temps linéaire est une technique classique de vérification formelle de systèmes concurrents. Un système, ainsi qu'une propriété qu'on souhaite y vérifier, sont modélisés sous forme d'omega-automates reconnaissant des mots infinis. Des manipulations de ces automates (produit synchronisé et test de vacuité) permettent d'établir si le système vérifie la propriété ou non.

Dans cette thèse nous nous focalisons sur un type particulier d'omega-automates qui permettent une représentation concise des propriétés d'équité faible : les automates de Büchi généralisés basés sur les transitions (TGBA ou *Transition-based Generalized Büchi Automata*).

Dans un premier temps, nous brossons un aperçu des algorithmes de vérification existant et nous en proposons de nouveaux traitant efficacement les automates généralisés forts. Dans un second temps, l'analyse des composantes fortement connexes de l'automate de la propriété nous a conduit à élaborer une décomposition de cet automate. Cette décomposition se focalise sur les automates multi-forces et permet une parallélisation naturelle des *model-checkers*. Enfin, nous avons proposé les premiers tests de vacuité parallèles pour les automates généralisés. De plus, tous ces tests sont lock-free à la différence de ceux de l'état de l'art. Toutes ces techniques ont ensuite été implémentées et évaluées sur un jeu de test conséquent.

Abstract

The automata-theoretic approach to linear time model-checking is a standard technique for formal verification of concurrent systems. The system and the property to check are modeled with omega-automata that recognizes infinite words. Operations overs these automata (synchronized product and emptiness checks) allows to determine whether the system satisfies the property or not.

In this thesis we focus on a particular type of omega-automata that enable a concise representation of weak fairness properties : transitions-based generalized Büchi automata (TGBA).

First we outline existing verification algorithms, and we propose new efficient algorithms for strong automata. In a second step, the analysis of the strongly connected components of the property automaton led us to develop a decomposition of this automata. This decomposition focuses on multi-strength property automata and allows a natural parallelization for already existing model-checkers. Finally, we proposed, for the first time, new parallel emptiness checks for generalized Büchi automata. Moreover, all these emptiness checks are lock-free, unlike those of the state-of-the-art. All these techniques have been implemented and then evaluated on a large benchmark.