

Thèse de doctorat de Sorbonne Université
Spécialité Informatique
Réalisée au sein du :
Laboratoire de Recherche de l'EPITA

Florian Renkin

Transformations d'ω-automates pour la synthèse de contrôleurs réactifs

Soutenue le 7 octobre 2022

Devant le jury composé de :

M. Olivier Carton — **Rapporteur**
Professeur des universités
Institut de Recherche en Informatique Fondamentale
Université de Paris Cité

M. Nicolas Markey — **Rapporteur**
Directeur de recherche, CNRS
Institut de Recherche en Informatique et Systèmes Aléatoires
Université de Rennes

M^me^ Hanna Klaudel — **Examinatrice**
Professeure des universités
Laboratoire Informatique, BioInformatique, Systèmes Complexes
Université d'Évry Paris-Saclay

M^me^ Laure Petrucci — **Examinatrice**
Professeure des universités
Laboratoire d'Informatique de Paris Nord
Université Sorbonne Paris Nord

M^me^ Nathalie Sznajder — **Examinatrice**
Maîtresse de Conférences
LIP6
Sorbonne Université

M. Alexandre Duret-Lutz — **Directeur de thèse**
Enseignant-Chercheur du privé
Laboratoire de Recherche de l'EPITA
EPITA

M. Adrien Pommellet — **Co-encadrant de thèse**
Enseignant-Chercheur du privé
Laboratoire de Recherche de l'EPITA
EPITA

Table des matières

1. Introduction

1.1. Motivation

De nos jours les systèmes dits réactifs sont partout autour de nous. On peut par exemple évoquer l'ampoule qui s'allume lorsque quelqu'un passe. Il associe un évènement qu'il observe (le passage d'une personne) à une action (l'allumage de l'ampoule). Créer un tel système est relativement simple mais surtout il s'agit d'un système qui peut rencontrer des problèmes sans que ce soit réellement grave. A contrario on peut considérer la gestion d'un feu tricolore. Laisser passer deux voies perpendiculaires peut causer des accidents. S'il s'agit encore d'un système simple, à la portée d'un raisonnement humain, des contraintes supplémentaires peuvent être ajoutées. Ainsi on peut ajouter un feu indépendant pour la voie de bus et imposer que les bus passent le moins de temps possible à attendre. Un moyen simple de faire cela est de laisser passer un bus dès qu'il arrive mais s'il y a toujours un bus qui souhaite passer ce feu, alors les voitures resteraient bloquées.

L'accumulation des contraintes fait qu'il devient difficile de concevoir un système qui réalise ce qui est demandé. De plus un humain peut faire des erreurs et il faut donc vérifier si ce qui a été produit est bien correct. Cette vérification est difficile pour un humain et c'est alors que se pose la question de l'utilisation de procédures automatiques. La plus courante est l'utilisation de tests. Cependant il est impossible de tester exhaustivement un programme complexe. Tester un système peut donner une preuve que le système est incorrect mais ne garantit pas l'absence de problème.

Les limites de ce système de vérification nous poussent à nous tourner vers d'autres méthodes. Ces méthodes dites formelles s'appuient sur le raisonnement autour d'un modèle et de propriétés à vérifier pour établir avec certitude que ce modèle vérifie bien ces propriétés.

Une première méthode qui est désignée par *vérification de modèle* ou *model checking* consiste à prendre un modèle décrivant les comportements du système à vérifier et à le comparer à une spécification formelle des comportements attendus.

Pour éviter de demander à un humain de faire un long travail qui sera ensuite vérifié par une machine, nous pouvons chercher une procédure qui, à partir d'un ensemble de propriétés à respecter, va fournir une machine correcte. C'est-à-dire une machine qui respecte la spécification et qui n'aura donc pas besoin d'être vérifiée.

Il convient de noter que cette méthode n'est pas parfaite. Le premier point à mettre en avant est qu'il faut une description précise du comportement que doit adopter le système. Ainsi si on demande un système qui allume une ampoule lorsqu'une présence est détectée, l'humain va comprendre qu'il faut éteindre la lumière lorsque personne n'est là. Lorsque ce système est créé automatiquement, la machine peut décider de créer un système laissant toujours l'ampoule allumée. Il y a donc un vrai travail pour obtenir une

description précise mais surtout comprise par une machine. Le second point à mettre en avant est que les méthodes automatisées (qu'il s'agisse de la vérification d'un système existant ou la création à partir de contraintes) s'appuient sur des modélisations. À partir d'un modèle représentant une solution il reste alors une étape consistant à transformer ce modèle en un système réel (programme, circuit...).

1.2. Contexte

Nous allons ici nous restreindre aux systèmes réactifs discrets. Un tel système prend en entrée une suite d'affectations de variables Booléennes (entrées) et produit une suite d'affectation de variables Booléennes de sortie.

De plus nous avons évoqué la spécification du comportement de l'interrupteur avec une phrase dans un langage naturel mais au-delà de son ambiguïté, ce n'est pas à la machine de chercher à comprendre nos mots. Une manière de décrire notre spécification peut être de passer par des formules logiques. Ainsi on peut spécifier le comportement de notre interrupteur avec "appui sur l'interrupteur ⇒ ampoule allumée".

Cependant comme nous travaillons sur des séquences d'entrées, nous avons parfois besoin de décrire des propriétés qui ne doivent être vraies qu'à certains moments. Pour revenir à notre exemple du feu tricolore, il nous faut une logique capable d'exprimer le fait qu'il doit toujours exister un moment où un feu donné passera au vert. C'est pour cela que nous utiliserons comme spécifications des formules de logique temporelle du temps linéaire (LTL). D'autres logiques, plus industrielles comme PSL [37] ou SVA [38] pourraient cependant être utilisées sans changer les méthodes décrites ici.

De même, parler de système est relativement vague. Un programme en Python est un système par exemple. Ici notre but sera de construire un circuit logique synchrone respectant la spécification.

Un concept important dans cette quête du système correct est l'ω-automate. Contrairement aux automates sur les mots finis, il va permettre de travailler sur des mots de taille infinie, ce qui est utile pour modéliser les comportements de notre système qui pour rappel travaille sur une suite (infinie) d'entrées.

1.3. Objectifs de la thèse

La création d'un système à partir d'une spécification est un problème pour lequel différentes approches ont été proposées, ce qui a conduit à l'apparition de divers outils dont certains seront étudiés dans ce manuscrit.

Le travail présenté ici se place dans le cadre de l'amélioration de `ltlsynt`, outil de Spot [21] dédié à la synthèse de contrôleurs réactifs à partir de spécifications LTL. L'amélioration concerne à la fois la rapidité de traitement mais aussi la qualité du résultat produit.

Alors que notre approche de la synthèse est composée de plusieurs étapes relativement indépendantes, nous allons nous concentrer sur deux d'entre elles.

Paritisation d'automate Les ω-automates possèdent une condition d'acceptation indiquant quelles exécutions sont valides. La paritisation consiste à transformer un ω-automate quelconque en un ω-automate avec une forme particulière de condition dite de parité.

Minimisation de machine de Mealy Une manière de décrire un contrôleur est l'utilisation d'une variante des machines de Mealy. Ces machines sont un moyen de faire correspondre à un mot de taille infinie un mot de taille infinie en sortie mais nous allons ici considérer un cadre plus général où il y a une possibilité pour une entrée de produire plusieurs mots de sortie. La minimisation de telles machines est maîtrisée mais implique une grande complexité. Le but est alors d'améliorer en pratique cette minimisation mais aussi d'introduire des techniques plus rapides, au prix d'un résultat possiblement non optimal.

Nous évoquerons également quelques méthodes spécifiques à certaines formules.

Découpage de formules LTL La première s'appuie sur la possibilité de diviser le travail en plusieurs sous-problèmes. Cette optimisation déjà décrite a été implémentée dans Spot et nous étudierons son efficacité.

Contournement de la construction usuelle Nous verrons que pour certaines formules, il est possible de se passer d'une partie des étapes usuelles pour construire une solution.

1.4. Plan

Le chapitre 2 donnera un aperçu de différents concepts que nous utilisons pour résoudre le problème de la synthèse LTL.

Le chapitre 3 introduira entre autres diverses notions de logique, de théorie des jeux mais nous parlerons surtout des ω-automates et des circuits dans le cadre de la synthèse LTL.

Le chapitre 4 sera l'occasion de montrer comment ces différentes notions sont utilisées pour produire un circuit à partir d'une spécification à travers notamment l'étude de divers outils.

Dans le chapitre 5, nous décrirons une première procédure de paritisation s'appuyant sur diverses méthodes existantes mais également de nouvelles heuristiques et transformations.

Le chapitre 6 introduira ACD, un algorithme postérieur à celui décrit dans le chapitre 5 qui donne en plus une garantie d'optimalité sur la taille du résultat.

Le chapitre 7 décrira deux manières de réduire la taille du contrôleur. Nous montrerons l'impact de ces réductions sur les stratégies obtenues lors de la SYNTCOMP mais aussi que notre modèle plus expressif que celui de l'outil référence (MeMin) s'insère mieux dans le cadre de la synthèse.

Le chapitre 8 sera l'occasion d'étudier différentes optimisations apportées au processus de synthèse. La première correspond au découpage de spécification de manière à travailler

sur des sous-systèmes plus simples. Le second est un nouveau processus de synthèse dédié à une sous-classe de formules permettant d'éviter de passer par les étapes usuelles.

2. Exemple

Ce chapitre a pour objectif de donner une vue d'ensemble du processus de synthèse que nous utilisons dans `ltlsynt` en s'appuyant sur l'exemple des feux de circulation. Il sera aussi l'occasion d'exprimer sous une forme plus formelle le comportement attendu.

2.1. Spécification

Dans le chapitre précédent nous posions le problème de la synthèse d'un circuit à partir d'une spécification. Un des problèmes évoqués était la création d'un feu de circulation qui ne laisse jamais une voiture attendre éternellement mais ne laisse pas deux feux au vert en même temps.

Modélisons le problème : nous considérons deux feux de circulation. Chaque feu i ne porte que deux couleurs : rouge et vert. À chaque feu i est associé un capteur C_i indiquant si au moins une voiture attend au feu correspondant.

Introduisons pour un feu i la variable R_i (resp. V_i) indiquant si l'ampoule rouge (resp. verte) du feu doit être allumée. De plus ajoutons aux opérateurs Booléens classiques l'opérateur unaire G précédant une formule devant toujours être vérifiée ainsi que F indiquant qu'une formule devra être vérifiée à un instant futur quelconque. Cela nous permet de décrire un ensemble de contraintes ainsi :

Description	Formule logique
À tout moment, au moins une des deux ampoules vertes est allumée	$\mathsf{G}(V_1 \vee V_2)$
Sur un même feu, l'ampoule rouge et l'ampoule verte ne peuvent jamais être allumées simultanément	$\bigwedge_{i\in\{1,2\}} \mathsf{G}(\neg(V_i \wedge R_i))$
Les feux ne peuvent jamais être de la même couleur	$\mathsf{G}((V_1 \leftrightarrow \neg V_2) \wedge (R_1 \leftrightarrow \neg R_2))$
Lorsque le capteur C_i détecte une voiture, l'ampoule associée à V_i devra s'allumer	$\bigwedge_{i\in\{1,2\}} \mathsf{G}(C_i \rightarrow \mathsf{F}(V_i))$

Remarque 1. *D'autres manières de décrire un feu peuvent être utilisées. Par exemple notre définition implique que les deux feux ne peuvent pas être rouges en même temps alors que ce comportement aurait pu être autorisé en pratique.*

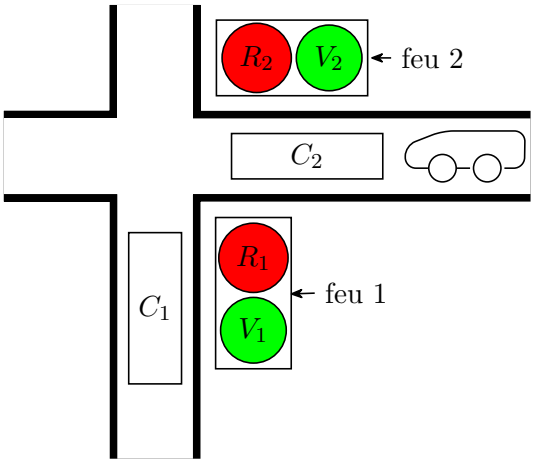


FIGURE 2.1. – Modélisation choisie d'un carrefour avec des feux de circulation. À chaque élément important est associé le nom de la variable Booléenne correspondante.

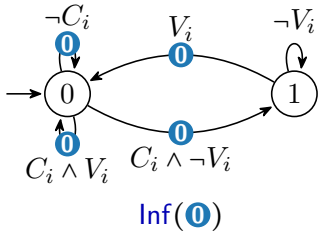


FIGURE 2.2. – Automate associé à $\mathsf{G}(C_i \to \mathsf{F}(V_i))$.

2.2. Utilisation d'automate

2.2.1. Représentation de la formule sous forme d'automate

Il nous faut alors un moyen de modéliser les propriétés décrites. Dans ces formules il n'existe qu'un nombre fini de variables qui ne peuvent prendre qu'un nombre fini de valeurs. Puisque l'état du système n'est décrit que par l'état de ces variables, alors il existe un nombre fini de configurations du système. Par exemple on peut être au moment où le feu 1 est vert mais que quelqu'un souhaiter passer au feu 2. Le passage d'une configuration à une autre se fait en fonction de la couleur des feux et de la présence de voitures devant chaque feu mais aussi de la configuration courante. Pour manipuler de tels systèmes nous utiliserons des automates sur les mots infinis.

Considérons la formule $\mathsf{G}(C_1 \to \mathsf{F}(V_1))$ indiquant que lorsqu'une voiture souhaite passer au feu 1, l'ampoule verte de ce feu finira par s'allumer. Un automate associé est

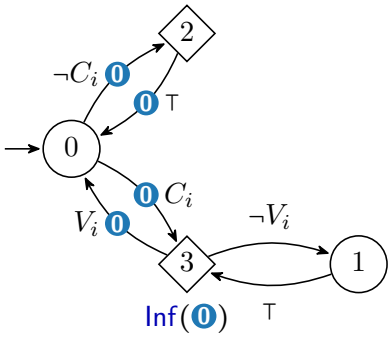


FIGURE 2.3. – Automate de la figure 2.2 découpé.

donné dans la figure 2.2. Ce dernier a deux états. Le premier (0) décrit que personne n'attend, c'est-à-dire qu'il n'y a personne ou que le feu est vert. Le second état (1) est la configuration où une voiture attend devant un feu qui n'est pas vert.

Pour passer de la première à la seconde configuration, il faut que le feu ne soit plus vert mais qu'une voiture arrive alors que pour faire le chemin inverse il faut que le feu passe au vert pour la laisser passer.

Cependant nous avons dans cet automate la possibilité de rester dans le second état, ce qui n'est pas ce que nous avons donné comme contrainte.

Ainsi à cet automate nous ajoutons l'obligation de voir infiniment souvent la pastille 0. Il est alors impossible d'être bloqué dans le second état et donc de ne pas laisser passer des voitures qui attendent.

2.2.2. Distinction des variables d'entrée et de sortie

Dans cet automate chaque arête porte des entrées et des sorties. Cependant, nous pouvons découper ces arêtes pour dissocier ces deux types de variable. L'idée est de d'abord prendre une arête qui voit des entrées puis une autre qui ne porte que des sorties. Nous obtenons alors l'automate de la figure 2.3. La forme des états y indique si les arêtes qui en sortent portent des entrées ou des sorties. Nous verrons cet automate comme un jeu où un joueur décrit comme l'*environnement* va jouer les signaux d'entrée alors que le *contrôleur* jouera les signaux de sortie.

2.2.3. Choix des arêtes de sortie

Principe

L'idée est ensuite de déterminer quelles arêtes de sortie doivent être utilisées pour que l'automate voit infiniment souvent la pastille 0. Ce choix ne peut porter que sur les arêtes portant des sorties.

Depuis l'état 2, peu importe la couleur du feu, 0 sera vue donc ce choix n'a pas d'importance. Depuis l'état 3, choisissons de ne pas allumer l'ampoule verte, ce qui nous

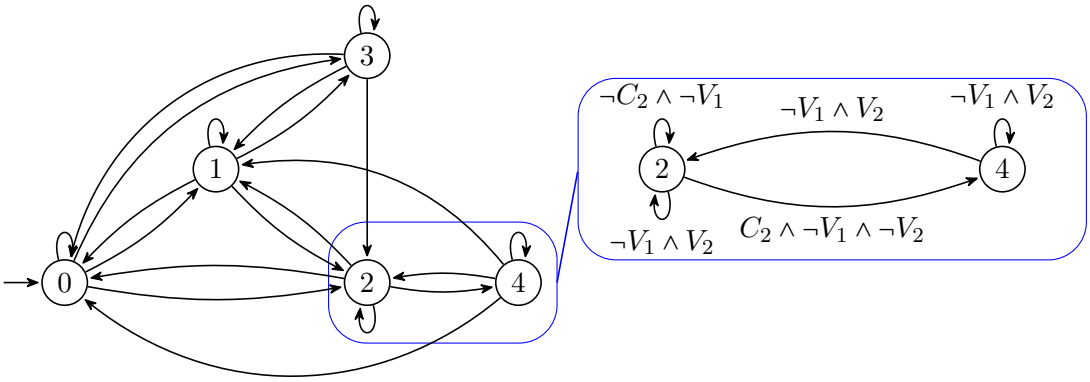


FIGURE 2.4. – Structure de l'automate associé à $\mathsf{G}(C_1 \to \mathsf{F}(V_1)) \wedge \mathsf{G}(C_2 \to \mathsf{F}(V_2))$. Les couleurs ainsi que les étiquettes y sont masquées.

amène à l'état 1. Depuis ce dernier, on ne peut aller que vers l'état 3. À partir de celui-ci, on allume l'ampoule verte et voit 0. Répéter ces choix va bien conduire à voir 0 infiniment souvent et donc respecter la contrainte donnée.

Il reste alors à décrire de manière plus concrète ce que doit faire le feu :

— Tant que personne n'attend, l'ampoule verte peut être allumée ou non ;

— Si quelqu'un attend, laisser l'ampoule verte éteinte puis l'allumer au moment suivant.

Existence de plusieurs choix

Nous avons choisi de ne pas allumer l'ampoule verte dès que quelqu'un attend. Cependant depuis l'état 3, il aurait été possible de l'allumer et le résultat aurait été correct. Le comportement aurait alors été plus simple :

— Tant que personne n'attend, l'ampoule verte peut être allumée ou non ;

— Si quelqu'un attend, allumer l'ampoule verte.

De même on peut restreindre ce qu'il est possible de faire lorsque personne n'attend en indiquant que le feu doit être vert. Le comportement est alors encore plus simple : toujours laisser le feu au vert.

2.2.4. Combinaison des spécifications

Nous n'avons considéré jusqu'à présent qu'une partie du problème, c'est-à-dire être capable de faire un contrôleur pour un seul feu qui ne laisse jamais attendre infiniment une voiture. Cependant, nous avons deux feux. Répétons alors en partie le même processus pour la formule LTL $\mathsf{G}(C_1 \to \mathsf{F}(V_1)) \wedge \mathsf{G}(C_2 \to \mathsf{F}(V_2))$. Pour construire cet automate, il est possible de faire le produit des automates associés aux deux sous-formules de cette conjonction. Même si le nombre d'états du résultat de ce produit est borné par le produit du nombre d'états de ces deux automates, nous verrons que le résultat de ce produit

doit être transformé (ce que l'on va nommer *paritisation*). Ainsi nous obtiendrons un automate avec au moins cinq états représenté figure 2.4.

Distinguer les entrées et les sorties nous donne un automate à 14 états et 35 arêtes. Il devient alors difficile à la main d'en extraire un comportement à suivre.

2.2.5. Découpage du problème

Considérons de nouveau la spécification $\mathsf{G}(C_1 \rightarrow \mathsf{F}(V_1)) \wedge \mathsf{G}(C_2 \rightarrow \mathsf{F}(V_2))$ indiquant que l'ampoule verte du feu 1 devra s'allumer si le capteur associé détecte quelqu'un et qu'il en est de même pour le second feu (sans empêcher que les deux ampoules vertes soient allumées en même temps).

Nous voyons dans la figure 2.4 qu'une arête (la boucle au-dessus de l'état 2) peut lier l'allumage d'une ampoule verte au capteur de l'autre feu. Ainsi travailler tel que nous l'avons décrit peut conduire à une spécification où l'ampoule verte d'un feu est liée au capteur de l'autre. Par exemple :

- Si le capteur 1 détecte une voiture, allumer les deux ampoules vertes
- Si le capteur 2 détecte une voiture, allumer l'ampoule verte du second feu

Ce comportement est valide puisqu'il finit toujours par laisser passer une voiture qui attend. Cependant est-il logique que l'allumage de la seconde ampoule dépende du capteur de l'autre feu ? On peut décider de découper le problème en deux parties et donc obtenir deux blocs de comportement :

- Feu 1 :
  - Tant que le capteur 1 ne voit personne, l'ampoule verte peut être allumée ou non,
  - Si le capteur 1 voit quelqu'un, laisser l'ampoule verte éteinte puis l'allumer au moment suivant,
- Feu 2 :
  - Allumer l'ampoule verte 2 si et seulement si le capteur 2 voit quelqu'un.

Ces deux feux ont alors un comportement différent (car nous l'avons décidé) et chaque ampoule verte ne dépend que de son capteur.

2.3. Conclusion

Nous avons transformé nos spécifications en un ensemble de formules logiques. Pour certaines d'entre elles, nous avons décrit comment nous pouvons en extraire un modèle.

Celui-ci peut être transformé et manipulé de manière à ce que des choix nous soient donnés. Ces choix nous permettent alors d'associer à chaque évènement un comportement que doivent adopter les variables de sortie pour que la spécification soit respectée. Cependant il n'existe pas un unique choix permettant ce respect de la spécification et en fonction de celui qui est choisi, il peut être plus ou moins simple de décrire le comportement à adopter.

Nous avons également évoqué la croissance rapide de la taille du modèle impliquant une grande difficulté à travailler sur des problèmes plus complexes.

3. Définitions

Le but de ce chapitre est de fournir au lecteur différentes notions qui lui seront utiles tout au long de ce manuscrit. Il s'agira de rappels sur les graphes mais aussi d'une introduction de la logique temporelle du temps linéaire. Nous introduirons également les ω-automates et montrerons leur lien avec la logique temporelle du temps linéaire. Des notions sur les jeux et la synthèse LTL seront également données.

3.1. Graphes

Nous allons ici décrire des notions importantes sur les graphes. Nous nous restreindrons aux graphes finis colorés orientés.

Définition 1 (Graphe). *Un graphe orienté fini est une paire $G = (V, E)$ où V est un ensemble fini de sommets et $E \subseteq V \times V$ est un ensemble d'arêtes.*

Définition 2 (Graphe coloré). *Un graphe coloré $G = (V, E, \Gamma)$ par un ensemble M est un graphe associé à une fonction $\Gamma : E \to 2^M$ colorant ses arêtes.*

Définition 3 (Sous-graphe). *Un graphe $G' = (V', E')$ est un sous-graphe d'un graphe G (noté $G' \subseteq G$) si $V' \subseteq V$ et $E' \subseteq E$.*

Définition 4 (Chemin). *Dans un graphe $G = (V, E)$, un chemin non-vide entre deux sommets u et v de V est une suite finie d'arêtes $((e_0, f_0), \ldots, (e_{n-1}, f_{n-1})) \in E^n$ telle que $e_0 = u$, $f_{n-1} = v$ et $\forall i \in \{0, n-2\}, f_i = e_{i+1}$.*

La longueur d'un chemin est le nombre d'arêtes (pas forcément distinctes) le composant.

Définition 5 (Graphe fortement connexe). *Un graphe $G = (V, E)$ est connexe si pour toute paire de sommets distincts $(u, v) \in V^2$ il existe un chemin de u à v ou un chemin de v à u. Un graphe est fortement connexe si pour toute paire de sommets $(u, v) \in V^2$ il existe un chemin de u à v.*

Définition 6 (Composante fortement connexe). *Une composante fortement connexe (SCC) d'un graphe G est un sous-graphe fortement connexe de G maximal au sens de l'inclusion.*

Définition 7 (SCC terminale). *Dans un graphe, une SCC est qualifiée de terminale si aucune autre SCC n'est accessible depuis un de ses sommets.*

Définition 8 (Cycle). *Un cycle d'un graphe $G = (V, E)$ est un chemin d'arêtes $((e_0, f_0), (e_1, f_1), \ldots, (e_{n-1}, f_{n-1})) \in E^n$ tel que $e_0 = f_{n-1}$.*

Définition 9 (Arbre). *Un arbre est un graphe orienté connexe sans cycle dans lequel tout sommet a une seule arête entrante sauf un unique sommet appelé racine qui n'en a pas. Ses sommets seront appelés nœuds.*

Définition 10 (Forêt). *Une forêt est un ensemble d'arbres.*

Définition 11 (Fils d'un nœud). *Dans un arbre $T = (V, E)$, un nœud v est un fils d'un nœud u si $(u, v) \in E$. On dit également que u est parent de v.*

L'ensemble des fils d'un nœud u est noté $Children_T$ et deux nœuds appartenant à un même tel ensemble sont qualifiés de frères.

Un nœud sans fils est une feuille ; l'ensemble des feuilles de T est désigné par $Leaves_T$.

Un nœud u est un descendant d'un nœud v s'il existe une suite finie de nœuds $(u_0, \dots, u_n)$ telle que $u_0 = u$, $u_n = v$ et pour tout $0 \leq i < n$, u_i est le parent de u_{i+1}.

Un nœud v est un ascendant d'un nœud u si u est un descendant de v.

Remarque 2. *Il est possible d'imposer un ordre partiel entre les nœuds d'un arbre.*

Définition 12 (Profondeur). *La profondeur d'un nœud u dans un arbre T est la longueur du chemin entre la racine de T et u et est notée $Depth_T(u)$ ou simplement $Depth(u)$ lorsqu'il ne peut pas y avoir de confusion.*

La hauteur de T est la plus grande profondeur d'un de ses nœuds (ou de manière équivalente de ses feuilles) et est notée $Height(T)$.

Définition 13 (Arbre étiqueté). *Soit A un ensemble d'étiquettes et $T = (V, E)$ un arbre. Un arbre A-étiqueté est une paire (T, η) comportant l'arbre et une fonction η associant à chaque état de V un élément de A.*

3.2. Logique temporelle du temps linéaire

Nous allons commencer par introduire l'algèbre de Boole avant de lui ajouter des opérateurs temporels afin de pouvoir créer des formules de logique temporelle du temps linéaire.

3.2.1. Algèbre de Boole

Notation 1 (Algèbre de Boole). *On note $\mathbb{B} = \{\top, \bot\}$ un ensemble contenant deux valeurs qui seront désignées par "vrai" et "faux".*

Les opérateurs Booléens suivants seront utilisés :

- *$\neg\phi$ pour la négation de ϕ ;*
- *$\phi_1 \vee \phi_2$ pour l'union de ϕ_1 et ϕ_2 ;*
- *$\phi_1 \wedge \phi_2$ pour l'intersection de ϕ_1 et ϕ_2, équivalent à $\neg(\neg\phi_1 \vee \neg\phi_2)$;*
- *$\phi_1 \to \phi_2$ pour l'implication de ϕ_2 par ϕ_1, équivalent à $\neg\phi_1 \vee \phi_2$;*
- *$\phi_1 \leftrightarrow \phi_2$ pour l'équivalence de ϕ_1 et ϕ_2, équivalent à $(\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$;*
- *$\phi_1 \oplus \phi_2$ pour l'union exclusive, équivalent à $(\neg\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \neg\phi_2)$;*

où ϕ_1, ϕ_2 et ϕ sont des formules Booléennes.

Définition 14 (Proposition atomique)**.** *On appelle proposition atomique une variable propositionnelle prenant ses valeurs dans $\mathbb{B}$.*

Notation 2. *Étant donné un ensemble P de propositions atomiques, on note $\mathbb{B}^P$ l'ensemble des valuations de P et $2^{\mathbb{B}^P}$ l'ensemble de ses sous-ensembles.*

Une valuation est notée comme un ensemble portant les propositions atomiques ou leur négation. Par exemple, pour $P = \{a, b\}$, une des valuations possibles est $\{a, \neg b\}$.

Une formule Booléenne peut être vue comme un ensemble de valuations de ses propositions atomiques. Par exemple, considérons la formule $a \vee b$ indiquant que a doit être vrai ou b doit être vrai. Cela correspond à l'ensemble des valuations $\{\{a, b\}, \{a, \neg b\}, \{\neg a, b\}\}$.

Notation 3. *Deux formules Booléennes ϕ_1 et ϕ_2 sont équivalentes et on note $\phi_1 \equiv \phi_2$ si elles représentent le même ensemble de valuations de propositions atomiques.*

Exemple 1 (Formule Booléenne)**.** *Décrivons à l'aide de deux propositions atomiques faim et manger que l'on mange si et seulement si l'on a faim. Cela correspond à la formule Booléenne $faim \leftrightarrow manger$.*

À l'aide des équivalences de la définition 1, on peut appliquer plusieurs règles de réécriture sur cette formule :

$$\begin{aligned} manger \leftrightarrow faim &\equiv (manger \rightarrow faim) \wedge (faim \rightarrow manger) \\ &\equiv (\neg manger \vee faim) \wedge (\neg faim \vee manger) \end{aligned}$$

Remarque 3. *Pour un ensemble P de propositions atomiques, on peut voir un élément de $\mathbb{B}^P$ comme une description de l'état d'un système. En effet, en reprenant le problème de l'exemple 1, $\{faim, manger\}$ indique que la personne étudiée a faim et est en train de manger.*

Plusieurs structures peuvent être utilisées pour représenter une fonction Booléenne comme les BDD. Nous utiliserons également une autre structure nommée cube.

Définition 15 (Cube)**.** *Un cube est une conjonction de littéraux.*

Exemple 2 (Cube)**.** *Étant données trois variables x_1, x_2 et x_3, le cube $x_1\bar{x_2}x_3$ signifie que x_1 et x_3 sont vraies alors que x_2 est fausse.*

Définition 16 (Forme normale disjonctive)**.** *Une formule Booléenne est sous forme normale disjonctive (FND) s'il s'agit d'une disjonction de conjonction de littéraux.*

Remarque 4. *Puisqu'une formule Booléenne peut toujours être réécrite en FND et qu'un cube est une conjonction de littéraux, alors toute fonction Booléenne peut être réécrite comme la disjonction de cubes.*

3.2.2. Formule LTL

En reprenant l'exemple précédent, on s'aperçoit qu'il n'est pas possible d'ajouter une notion de temps. Par exemple il peut être utile de dire que si on a faim, il existera un moment où l'on va manger. Nous n'allons travailler que sur un *temps discret*, c'est-à-dire que la variable décrivant un instant est discrète (pour tout intervalle borné, il n'existe qu'un nombre fini de valeurs dans cet intervalle).

Définitions

Pour ajouter cette notion de temporalité, nous ajoutons à l'algèbre de Boole l'ensemble des opérateurs suivants :

$\mathsf{G}\phi$ indiquant que la valuation des variables de ϕ satisfait toujours ϕ.

$\mathsf{F}\phi$ indiquant qu'il existera un moment où la valuation des variables de ϕ satisfera ϕ.

$\mathsf{X}\phi$ indiquant que la prochaine valuation des variables de ϕ satisfera ϕ.

$\phi_1 \mathsf{U} \phi_2$ indiquant que tant que la valuation ne satisfait pas ϕ_2 elle doit satisfaire ϕ_1 et ϕ_2 sera vrai.

$\phi_1 \mathsf{W} \phi_2$ indiquant que tant que la valuation ne satisfait pas ϕ_2 elle doit satisfaire ϕ_1 et devra toujours satisfaire ϕ_1 si ϕ_2 n'est jamais satisfait.

$\phi_1 \mathsf{M} \phi_2$ indiquant ϕ_2 doit être vérifié jusqu'à ce que $\phi_1 \wedge \phi_2$ soit vérifié et $\phi_1 \wedge \phi_2$ sera vérifié.

$\phi_1 \mathsf{R} \phi_2$ indiquant ϕ_2 doit être vérifié jusqu'à ce que $\phi_1 \wedge \phi_2$ soit vérifié et ϕ_2 sera toujours vérifié si $\phi_1 \wedge \phi_2$ ne l'est jamais.

Définition 17 (Formule LTL, [63])**.** *Une formule LTL est définie de manière récursive par :*

- *Une proposition atomique est une formule LTL ;*
- *Si ϕ_1 et ϕ_2 sont des formules LTL, alors*
  - *$\nabla\phi_1$ est une formule LTL pour $\nabla \in \{\mathsf{G}, \mathsf{F}, \mathsf{X}\}$,*
  - *$\neg\phi_1$ est une formule LTL,*
  - *$\phi_1 \triangleright \phi_2$ est une formule LTL pour $\triangleright \in \{\vee, \wedge\}$,*
  - *$\phi_1 \triangle \phi_2$ est une formule LTL pour $\triangle \in \{\mathsf{U}, \mathsf{M}, \mathsf{R}, \mathsf{W}\}$.*

Les formules LTL sur un ensemble P de propositions atomiques permettent de décrire des ensembles de suites infinies dans $2^{\mathbb{B}^P}$.

Définition 18 (Sémantique d'une formule LTL)**.** *Soient P un ensemble de propositions atomiques et $w = w_1 w_2 \ldots \in 2^{\mathbb{B}^P}$ une suite infinie [1] de valuations sur P. On note $w_{i..}$ le suffixe de w commençant à la position i. La relation de satisfaction $\vDash_{LTL}$ est définie par :*

- *$w \vDash_{LTL} p \in P$ si $p \in w_1$;*

1. Dans le cas des suites finies, on se tournera vers les formules LTLf [19]

— $w \models_{LTL} \neg\phi$ *si* $w \not\models_{LTL} \phi$;
— $w \models_{LTL} \mathsf{X}\phi$ *si* $w_{2..} \models_{LTL} \phi$;
— $w \models_{LTL} \mathsf{G}\phi$ *si* $\forall i, w_{i..} \models_{LTL} \phi$;
— $w \models_{LTL} \mathsf{F}\phi$ *si* $\exists i, w_{i..} \models_{LTL} \phi$;
— $w \models_{LTL} \phi\mathsf{U}\psi$ *si* $\exists j, \forall i < j, w_{i..} \models_{LTL} \phi$ *et* $w_{j..} \models_{LTL} \psi$;
— $w \models_{LTL} \phi\mathsf{W}\psi$ *si* $(w \models_{LTL} \phi\mathsf{U}\psi) \vee (w \models_{LTL} \mathsf{G}\phi)$;
— $w \models_{LTL} \phi\mathsf{M}\psi$ *si* $\exists j, \forall i \leq j, w_{i..} \models_{LTL} \psi$ *et* $w_{j..} \models_{LTL} \phi$;
— $w \models_{LTL} \phi\mathsf{R}\psi$ *si* $(w \models_{LTL} \phi\mathsf{M}\psi) \vee (w \models_{LTL} \mathsf{G}\psi)$.

3.3. ω-automate

Un *ω-automate* (ou automate sur les mots de longueur infinie) est un automate reconnaissant des langages où les mots sont de taille infinie.

Contrairement aux automates sur les mots finis il n'est pas possible de décider de l'acceptation d'un mot en étudiant dans quel état termine l'exécution puisqu'elle est infinie.

Plusieurs types d'ω-automates existent mais ceux que nous considérerons sont constituées d'un ensemble d'états et décident de l'acceptation d'une exécution en fonction d'éléments qui sont vus finiment souvent ou non. Ces automates diffèrent globalement de la manière dont est décrite cette condition mais aussi des éléments qui sont étudiés (états, couleurs...).

Dans cette section nous allons définir ces automates et les classifierons selon différentes classes que nous comparerons.

3.3.1. Automate à acceptation décrite par des ensembles d'états

Le premier type d'ω-automate qui sera décrit sera l'automate de Muller. Il s'agit d'un automate où l'acceptation est décidée en fonction des états visités infiniment souvent. Une exécution y est acceptante si l'ensemble des états visités infiniment souvent correspond exactement à un des ensembles décrit par la condition d'acceptation.

Définition 19 (Automate de Muller)**.** *Un automate de* Muller *est un tuple* $(Q, \Sigma, \delta, q_0, T)$ *où*

— Q *est un ensemble fini d'états ;*
— Σ *est un alphabet fini d'entrée ;*
— $\delta \subseteq Q \times \Sigma \times Q$ *est un ensemble de transitions ;*
— q_0 *est un état initial ;*
— $T \subseteq 2^{2^Q}$ *est un ensemble d'ensembles d'états acceptants.*

Notation 4. *Pour une transition* $d = (q_1, \ell, q_2) \in \delta$*, on écrit* $q \xrightarrow{\ell} q_2$.

$\{0,1\},\{0,2\}$ $\qquad$ $\{0\},\{0,1\},\{0,2\},\{0,1,2\}$

(a) $\qquad$ (b)

FIGURE 3.1. – Automates de Muller sur $\Sigma = \{a,b\}$

Définition 20 (Exécution d'un automate de Muller)**.** *Une exécution r d'un automate de Muller $\mathcal{A} = (Q,\Sigma,\delta,q_0,T)$ est une suite infinie de transitions $r = (s_i \xrightarrow{\ell_i} s'_i)_{i\in\mathbb{N}} \in \delta^\omega$ telle que $s_0 = q_0$ et $\forall i \geq 0$, $s'_i = s_{i+1}$.*

Puisqu'il existe un nombre fini d'éléments dans Q, alors il existe un rang $j_r \geq 0$ tel que pour tout $i \geq j_r$, l'état s_i est vu infiniment souvent dans r.

Définition 21 (Sémantique d'un automate de Muller)**.** *Soit $\mathcal{A}$, r et j_r tels que définis précédemment. Notons $\mathsf{Rep}(r) = \bigcup_{i\geq j_r} s_i$ l'ensemble des états vus infiniment souvent dans r.*

L'exécution r est acceptante si $\mathsf{Rep}(r) \in T$ et on dit alors que $\mathcal{A}$ accepte *le mot $(\ell_i)_{i\geq 0} \in \Sigma^\omega$.*

Définition 22 (Langage reconnu par un automate)**.** *Le langage reconnu par un automate $\mathcal{A}$ donné est l'ensemble des mots acceptés par $\mathcal{A}$ et est noté $\mathcal{L}(\mathcal{A})$.*

Exemple 3. *Considérons l'automate de Muller $\mathcal{A}$ de la figure 3.1a. La condition d'acceptation indique que l'ensemble des états visités infiniment souvent doit être $\{0,1\}$ ou $\{0,2\}$.*

Ainsi l'idée est qu'un mot est accepté si et seulement si à partir d'un certain moment, il ne voit plus qu'une des deux lettres. En effet, l'état 1 indique que la dernière lettre vue est a alors que l'état 2 indique que la dernière lettre vue est b.

Le langage $\mathcal{L}(\mathcal{A})$ associé à cet automate est donc l'ensemble des mots qui contiennent un nombre quelconque fini d'éléments de $\Sigma = \{a,b\}$ puis une suite infinie de a ou une suite infinie de b.

Même si nous n'allons pas nous servir de cette notation, nous pouvons noter que ce langage peut être décrit comme $\Sigma^*(a^\omega + b^\omega)$ où l'étoile à la même signification que dans le cas des mots finis alors que l'exposant ω indique la répétition infinie.

Le deuxième type d'automate que nous allons décrire s'appuie sur l'automate de la figure 3.1b dont les étiquettes sont masquées. La condition qui y est utilisée indique que

tout ensemble acceptant est un sur-ensemble de $\{0\}$. La condition est donc équivalente à voir infiniment souvent l'état 0, sans prendre en compte les autres états.

L'idée est alors d'avoir une condition indiquant que les états vus infiniment souvent doivent intersecter l'ensemble des états décrits par la condition (ici $\{0\}$).

Ce type d'automate est appelé automate de Büchi et est défini de la manière suivante :

Définition 23 (Automate de Büchi)**.** *Un automate de Büchi est un tuple* $(Q, \Sigma, \delta, q_0, T)$ *où*

- Q *est un ensemble fini d'états ;*
- Σ *est un alphabet fini d'entrée ;*
- $\delta \subseteq Q \times \Sigma \times Q$ *est un ensemble de transitions ;*
- q_0 *est un état initial ;*
- $T \subseteq Q$ *est un ensemble d'états acceptants.*

La sémantique de l'automate ne diffère des automates de Muller que par la non-vacuité de l'intersection de Rep et T :

Définition 24 (Sémantique d'un automate de Büchi)**.** *Une exécution r est acceptante si et seulement si* $\mathsf{Rep}(r) \cap T \neq \varnothing$.

Remarque 5. *Passer d'une condition de Büchi à une condition de Muller consiste à décrire tous les sur-ensembles des sous-ensembles de T, c'est-à-dire* $\{P \subseteq Q \mid P \cap T \neq \varnothing\}$.

Théorème 1 (Différence d'expressivité des automates de Büchi et de Muller [33, p. 12])**.** *Il existe des automates de Muller pour lesquels il est impossible d'avoir un automate de Büchi déterministe équivalent.*

Alors que les automates de Büchi déterministes ne sont pas aussi expressifs que les automates de Muller, d'autres types de conditions impliquent une conservation de l'expressivité.

Définition 25 (Automate de Rabin)**.** *Les automates de Rabin et les automates de Streett sont des tuples* $(Q, \Sigma, \delta, q_0, \alpha)$ *où*

- Q *est un ensemble fini d'états ;*
- Σ *est un alphabet fini d'entrée ;*
- $\delta \subseteq Q \times \Sigma \times Q$ *est un ensemble de transitions ;*
- q_0 *est un état initial ;*
- $\alpha = \{(E_1, F_1), \ldots, (E_n, F_n)\}$ *est un ensemble de paires d'ensembles d'états.*

Définition 26 (Sémantique d'un automate de Rabin)**.** *Étant donné un automate de Rabin $\mathcal{A}$ portant une condition α, une exécution r sur $\mathcal{A}$ est acceptante s'il existe une paire (E_i, F_i) telle que des états de F_i sont vus infiniment souvent et ceux de E_i sont vus finiment souvent.*

Définition 27 (Sémantique d'un automate de Streett)**.** *Étant donné un automate de Streett $\mathcal{A}$ portant une condition α, une exécution r sur $\mathcal{A}$ est acceptante si pour toute paire (E_i, F_i) soit E_i contient des états vus infiniment souvent ou F_i ne contient pas d'état vu infiniment souvent.*

Définition 28 (Automate de parité)**.** *Un automate de parité [2] est un automate de Rabin dont la condition $\{(E_1, F_1), \ldots, (E_n, F_n)\}$ vérifie $E_1 \subseteq F_1 \subseteq \ldots \subseteq E_n \subseteq F_n$.*

En dehors de la condition associée à un automate, on peut également tirer d'autres propriétés de ces automates.

Définition 29 (Automate faible)**.** *Un automate $\mathcal{A}$ est faible si toute SCC ne rejette aucun mot ou n'accepte aucun mot.*

Définition 30 (Automate terminal)**.** *Un automate $\mathcal{A}$ est terminal s'il est faible et qu'une SCC rejetante n'est pas accessible depuis une SCC acceptante. De plus, toute SCC acceptante est complète.*

3.3.2. Condition d'Emerson-Lei

Nous avons vu que les conditions d'acceptation des ω-automates peuvent s'exprimer comme une description d'ensembles d'états qui doivent être visités infiniment souvent avec les conditions de Muller et de Büchi. Les conditions de Streett et de Rabin (et de parité de manière plus abstraite) nous ont également permis de décrire certains ensembles d'états qui ne doivent pas être vus infiniment souvent.

Cependant nous allons ici colorier les états des automates et faire porter la condition sur cette couleur. Ainsi au lieu de décrire une condition de Büchi comme un ensemble d'états, on peut se limiter à simplement demander à voir infiniment souvent une couleur.

De plus au lieu de décrire une condition comme une énumération des ensembles d'états (ou couleurs) qui doivent être visités infiniment souvent, nous allons ici décrire les conditions à l'aide d'un formule Booléenne. Ainsi $\{\{0\}, \{1\}\}$ peut être décrit comme l'obligation de voir finiment souvent 0 ou finiment souvent 1.

Remarque 6. *Puisqu'une même couleur peut être partagée entre plusieurs états, cela conduit à une condition plus compacte sans perte d'expressivité.*

Emerson et Lei définirent [22] un type de condition d'acceptation que Safra et Vardi nommèrent condition d'Emerson-Lei. À la suite de l'émergence du format HOA [5] [3], ce type de condition a été popularisé. Il s'appuie sur deux opérateurs unaires. Le premier est Inf et permet d'indiquer qu'une couleur devra être vue infiniment souvent. À l'inverse, l'opérateur Fin indique qu'une couleur ne devra pas être vue infiniment souvent.

2. Carton nomme de tels automates *chain automata* [12]. Cependant il s'agit d'une condition équivalente à ce que nous définirons comme des conditions de parité pour les conditions d'Emerson-Lei.

3. HOA pour Hanoi Omega-Automata, dont le nom fait référence à Hanoï, lieu où les discussions sur ce format ont commencées.

Plus formellement, l'ensemble $\mathcal{C}(M)$ des *conditions d'acceptation d'Emerson-Lei* correspond aux formules respectant la grammaire suivante où m est un élément quelconque de M :

$$\alpha ::= \top \mid \bot \mid \mathsf{Inf}(m) \mid \mathsf{Fin}(m) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha)$$

Pour un ensemble de couleurs $N \subseteq M$ nous définissons la relation $N \vDash \alpha$ à partir de la sémantique suivante :

$N \vDash \top$	$N \vDash \mathsf{Inf}(m)$ iff $m \in N$	$N \vDash (\alpha_1 \wedge \alpha_2)$ iff $N \vDash \alpha_1$ et $N \vDash \alpha_2$
$N \nvDash \bot$	$N \vDash \mathsf{Fin}(m)$ iff $m \notin N$	$N \vDash (\alpha_1 \vee \alpha_2)$ iff $N \vDash \alpha_1$ ou $N \vDash \alpha_2$

Remarque 7. *Puisque* Fin *correspond à la négation de* Inf*, et que l'implication et l'équivalence ne sont que du sucre syntaxique, la grammaire de $\mathcal{C}$ est suffisante pour décrire tous les sous-ensembles de couleurs possibles.*

Par exemple, $\mathsf{Inf}(0) \rightarrow \mathsf{Inf}(1)$ *est équivalent à* $\neg\mathsf{Inf}(0) \vee \mathsf{Inf}(1)$ *lui-même équivalent à* $\mathsf{Fin}(0) \vee \mathsf{Inf}(1)$.

Définition 31 (Dual d'une condition)**.** *Le dual d'une condition d'Emerson-Lei α est la formule résultant de la permutation des* Fin *et des* Inf *ainsi que des $\vee$ et des $\wedge$. Elle correspond à la négation de α et est notée $\neg\alpha$.*

Exemple 4 (Dual d'une condition de Rabin)**.** *Le dual de la condition de Rabin* $(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(3))$ *est la condition de Streett* $(\mathsf{Inf}(0) \vee \mathsf{Fin}(1)) \wedge (\mathsf{Inf}(2) \vee \mathsf{Fin}(3))$.

Classes de formules

Même si une condition d'Emerson-Lei peut être une formule positive quelconque, il est possible de mettre en avant certaines ayant une structure particulière. Celles-ci sont décrites dans le tableau 3.1.

Définition 32 (Paire de Rabin et Streett)**.** *Étant donnée une condition de Rabin (resp. de Streett), on appelle* paire de Rabin *(resp.* paire de Streett*) chacune des clauses de cette condition.*

La couleur associée à un Inf *sera décrite comme la* couleur requise *alors que celle associée au* Fin *sera décrite comme la* couleur interdite*.*

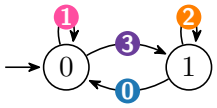
Remarque 8. *Il est possible de transformer un automate portant une condition Rabin-like* $\bigvee_i (\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1})) \vee \bigvee_j \mathsf{Inf}(m_j) \vee \bigvee_k \mathsf{Fin}(m_k)$ *en automate portant une condition de Rabin* $\bigvee_i \big(\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1})\big) \vee \bigvee_j (\mathsf{Fin}(a) \wedge \mathsf{Inf}(m_j)) \vee \bigvee_k (\mathsf{Fin}(m_k) \wedge \mathsf{Inf}(b))$ *en introduisant deux nouvelles couleurs a et b telles que a n'apparaît pas dans l'automate alors que b est partout. Cela signifie qu'un algorithme adapté aux automates de Rabin ou Streett est applicable aux automates Rabin-like ou Streett-like.*

Remarque 9. *Une condition comme* $\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)$ *peut être vue comme une condition de Rabin à une paire ou une condition Streett-like à deux paires. De manière similaire, une condition de Büchi généralisé peut être vue comme une condition Streett-like.*

TABLE 3.1. – Structure des conditions d'acceptation usuelles. Les variables $m, m_0, m_1, \ldots, m_i^j$ désignent n'importe quelle couleur dans $M = \{0, 1, \ldots\}$ pour autoriser de multiples occurrences.

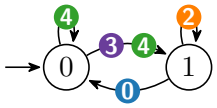
Büchi	$\mathsf{Inf}(m)$ L'unique couleur doit être vue infiniment souvent
Büchi généralisé	$\bigwedge_i \mathsf{Inf}(m_i)$ Toutes les couleurs doivent être vues infiniment souvent
co-Büchi	$\mathsf{Fin}(m)$ L'unique couleur doit être vue finiment souvent
co-Büchi généralisé	$\bigvee_i \mathsf{Fin}(m_i)$ Au moins une des couleurs ne doit pas être vue infiniment souvent
Rabin	$\bigvee_i (\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1}))$ Les couleurs vues finiment et infiniment souvent doivent correspondre à au moins une des paires
Rabin-like	$\bigvee_i (\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1})) \vee \bigvee_j \mathsf{Inf}(m_j) \vee \bigvee_k \mathsf{Fin}(m_k)$ Une condition de Rabin où certaines paires peuvent ne pas porter de Inf ou de Fin
Rabin généralisée	$\bigvee_i \mathsf{Fin}(m_i) \wedge \mathsf{Inf}(m_i^1) \wedge \mathsf{Inf}(m_i^2) \ldots \wedge \mathsf{Inf}(m_i^k)$ Condition de Rabin où la partie Inf d'une paire est remplacée par une conjonction de Inf
Streett	$\bigwedge_i (\mathsf{Inf}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1}))$ Chaque paire doit décrire une couleur qui est vue infiniment ou une couleur vue finiment souvent
Streett-like	$\bigwedge_i (\mathsf{Inf}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1})) \wedge \bigwedge_j \mathsf{Inf}(m_j) \wedge \bigwedge_k \mathsf{Fin}(m_k)$ Condition de Streett où une paire peut ne pas porter de Inf ou de Fin
parité maximale paire	$\mathsf{Inf}(2k) \vee (\mathsf{Fin}(2k-1) \wedge (\mathsf{Inf}(2k-2) \vee (\mathsf{Fin}(2k-3) \wedge \ldots)))$ Voit une couleur comme un entier pour indiquer que la plus grande couleur vue infiniment souvent doit être paire
parité maximale impaire	$\mathsf{Inf}(2k+1) \vee (\mathsf{Fin}(2k) \wedge (\mathsf{Inf}(2k-1) \vee (\mathsf{Fin}(2k-2) \wedge \ldots)))$ Voit une couleur comme un entier pour indiquer que la plus grande couleur vue infiniment souvent doit être impaire
parité minimale paire	$\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge (\mathsf{Inf}(2) \vee \ldots))$ Voit une couleur comme un entier pour indiquer que la plus petite couleur vue infiniment souvent doit être paire

Remarque 10. *Toute sous-condition de la forme* $\bigvee_i \mathsf{Inf}(m_i)$ *(resp.* $\bigwedge_i \mathsf{Fin}(m_i)$*) peut être réécrite comme* $\mathsf{Inf}(a)$ *(resp.* $\mathsf{Fin}(a)$*) en introduisant une couleur* a *sur toutes les arêtes sur lesquelles un des* m_i *apparaît. Ainsi toute condition de parité peut être réécrite comme une condition Rabin-like (resp. Streett-like) en la mettant sous forme normale disjonctive (resp. conjonctive) puis en remplaçant chaque terme de la forme* $\bigwedge_i \mathsf{Fin}(m_i)$



$\mathsf{Fin}(3) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0)))$

(a) Automate de parité



$(\mathsf{Fin}(4) \wedge \mathsf{Inf}(0)) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(2))$

(b) Automate Rabin-like équivalent à celui de la figure 3.2a

FIGURE 3.2. – Transformation d'un automate de parité en automate Rabin-like

(resp. $\bigvee_i \mathsf{Inf}(m_i)$*) par un unique* Fin *(resp.* Inf*).*

Exemple 5 (Réécriture d'une condition de parité en condition Rabin-like)**.** *Appuyons-nous sur l'automate de la figure 3.2a portant la condition* $\mathsf{Fin}(3) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0)))$*. Cette condition de parité est équivalente à* $(\mathsf{Fin}(1) \wedge \mathsf{Fin}(3) \wedge \mathsf{Inf}(0)) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(2))$ *qui peut être simplifiée en* $(\mathsf{Fin}(4) \wedge \mathsf{Inf}(0)) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(2))$ *où* 4 *est une nouvelle couleur présente sur toute arête portant* 1 *ou* 3*. Puisque* 1 *n'est plus présente dans la condition, alors elle peut être supprimée de l'automate. Le résultat est décrit dans la figure 3.2b.*

Remarque 11. *Puisqu'une condition d'Emerson-Lei est une formule Booléenne, il est possible de l'écrire sous forme normale disjonctive. Ainsi, la condition* $((\mathsf{Fin}(0) \vee \mathsf{Fin}(1)) \wedge \mathsf{Inf}(2)) \vee \mathsf{Fin}(3)$ *est équivalente à* $(\mathsf{Fin}(0) \wedge \mathsf{Inf}(2)) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(2)) \vee \mathsf{Fin}(3)$ *qui est une condition de Rabin généralisée. Pour obtenir une condition de Rabin généralisée, il peut y avoir besoin de fusionner des* Fin *(voir remarque 10).*

De manière analogue, réécrire une formule sous forme normale conjonctive conduit à l'obtention d'une condition de Streett généralisée.

Remarque 12. *Pour passer d'un automate de Muller à un automate d'Emerson-Lei, il suffit d'attribuer à chaque état une couleur différente puis de décrire les exécutions acceptantes à partir de ces couleurs.*

Exemple 6 (Transformation d'un automate de Muller en automate d'Emerson-Lei)**.** *Considérons l'automate de Muller de la figure 3.1a. Colorons l'état 0 par* 0*, 1 par* 1 *et 2 par* 2*. Si on utilise la condition* $(\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \wedge \mathsf{Fin}(2)) \vee (\mathsf{Inf}(0) \wedge \mathsf{Inf}(2) \wedge \mathsf{Fin}(1))$ *avec cette coloration, alors nous obtenons un automate d'Emerson-Lei équivalent à l'automate de Muller donné.*

Une utilité de cette classification est la possibilité de spécialiser des algorithmes. On peut par exemple évoquer le test de vacuité d'un automate donné.

De manière générale, il s'agit de trouver un cycle portant un ensemble de couleur satisfaisant la condition d'acceptation. Dans le cas d'un automate de Büchi, ce test se résume à la recherche d'une arête colorée accessible dans un cycle, qui est faisable

condition		formule	interprétation alternative
maximale impaire	0	$\top$	accepte tout
	1	$\mathsf{Fin}(0)$	co-Büchi
	2	$\mathsf{Inf}(1)\vee\mathsf{Fin}(0)$	Streett(1), Rabin-like(2)
	3	$\mathsf{Fin}(2)\wedge(\mathsf{Inf}(1)\vee\mathsf{Fin}(0))$	Streett-like(2)
	4	$\mathsf{Inf}(3)\vee(\mathsf{Fin}(2)\wedge(\mathsf{Inf}(1)\vee\mathsf{Fin}(0)))$	
	5	$\mathsf{Fin}(4)\wedge(\mathsf{Inf}(3)\vee(\mathsf{Fin}(2)\wedge(\mathsf{Inf}(1)\vee\mathsf{Fin}(0))))$	
maximale paire	0	$\bot$	rejette tout
	1	$\mathsf{Inf}(0)$	Büchi
	2	$\mathsf{Fin}(1)\wedge\mathsf{Inf}(0)$	Rabin(1), Streett-like(2)
	3	$\mathsf{Inf}(2)\vee(\mathsf{Fin}(1)\wedge\mathsf{Inf}(0))$	Rabin-like(2)
	4	$\mathsf{Fin}(3)\wedge(\mathsf{Inf}(2)\vee(\mathsf{Fin}(1)\wedge\mathsf{Inf}(0)))$	
	5	$\mathsf{Inf}(4)\vee(\mathsf{Fin}(3)\wedge(\mathsf{Inf}(2)\vee(\mathsf{Fin}(1)\wedge\mathsf{Inf}(0))))$	

TABLE 3.2. – Exemples de conditions de parité

avec un algorithme de recherche de composantes fortement connexe et est donc linéaire en la taille de l'automate. A l'inverse pour une condition quelconque, le problème est NP-complet [22].

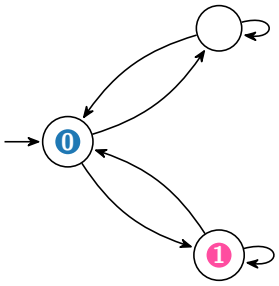
Certaines propriétés peuvent aussi être liées au type de la condition associée à un automate. Par exemple dans un automate de Streett l'union de deux cycles rejetants est rejetante.

Remarque 13. *La table 3.2 décrit les conditions de parité maximale impaire et paire pour plusieurs nombres de couleurs en utilisant la syntaxe du format HOA [5]. Cela permet de voir que dans certains cas une formule peut être interprétée de différentes manières (principalement lorsqu'il y a au plus une couleur). Dans le cas des conditions de Rabin(-like) et Streett(-like), le nombre de paires est indiqué entre parenthèses (voir la remarque 9).*

Remarque 14. *L'existence d'élément non-coloré dans ces automates est possible. On peut voir cette absence de couleur comme un coloriage par une couleur imaginaire* $\textcircled{-1}$*. Ainsi pour des applications ayant besoin d'un automate où chaque élément porte une couleur, il est possible d'ajouter 1 à toutes les couleurs de l'automate et de passer d'une condition de parité maximale paire à maximale impaire ou vice-versa.*

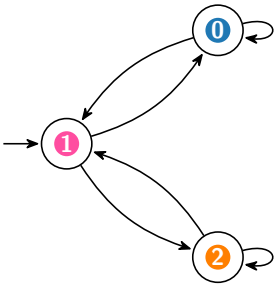
Exemple 7. *Dans la figure 3.3a nous avons un automate où il y a un état qui n'est pas colorié. Puisqu'il possède une couleur* $\textcircled{-1}$*, augmenter sa valeur fait qu'il est colorié par* 0*. De la même manière la couleur associée aux deux autres états est incrémentée. Enfin la condition est adaptée de manière à ce que l'automate de la figure 3.3b soit bien équivalent à celui de la figure 3.3a.*

Il est possible de vérifier rapidement que ces deux automates sont bien équivalents. Dans le premier, on peut aller dans l'état colorié avec 1 *infiniment souvent et l'exécution sera*



$\mathsf{Inf}(1) \vee \mathsf{Fin}(0)$

(a) Automate de parité maximale impaire partiellement colorié



$\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))$

(b) Automate de parité maximale paire totalement colorié

FIGURE 3.3. – Transformation d'un automate de parité de manière n'avoir que des états colorés.

acceptante. Par contre si on visite infiniment souvent l'état non-colorié mais finiment celui colorié avec 1*, il ne faut jamais visiter l'état colorié par* 0 *pour que l'exécution soit acceptante.*

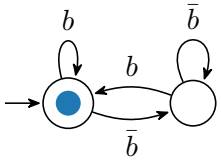
Dans le second cas, le passage infiniment souvent par 2 *engendre bien l'acceptation de l'exécution alors qu'il est toujours nécessaire de passer infiniment souvent par l'état colorié par* 1 *si celui colorié par* 2 *n'est pas visité infiniment souvent.*

Remarque 15. *Il est facile de passer d'une condition de parité minimale à une condition de parité maximale ou inversement. En effet, supposons que toutes les arêtes sont colorées. Il suffit alors d'inverser les couleurs, avec un décalage de 1 si besoin. Par exemple,* $\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(2))$ *peut devenir* $\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))$ *en permutant* 0 *et* 2 *dans la condition et l'automate.*

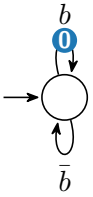
Une autre utilité de ce découpage est la possibilité d'obtenir un automate plus compact en changeant la condition d'acceptation [9]. Comme nous le prouverons page 61, transformer un automate déterministe ayant n états et portant une condition de Rabin à k paires en automate de parité peut conduire à un automate ayant $n \cdot k!$ états.

3.3.3. Automates de Büchi généralisés

Dans le domaine de la vérification de modèles, la complémentation et l'intersection d'automates sont des opérations cruciales. En effet, il s'agit globalement de modéliser un système et la négation de spécifications qu'il doit respecter puis de vérifier que ces deux modèles n'ont pas de comportement commun. Il est alors préférable de manipuler des automates de Büchi déterministes puisque ces opérations produisent des automates de taille respectivement quadratique et identique lorsque cela est possible [9]. Il faut cependant mettre en avant que certaines formules LTL ne peuvent pas être décrites par un automate de Büchi et dans le cas non-déterministe, ces opérations sont respectivement exponentielles et quadratiques.



(a) Automate de Büchi sur les états pour GF(b)



(b) Automate de Büchi pour GF(b) où les couleurs sont portées par les transitions

FIGURE 3.4. – Automates de Büchi associés à GF(b)

Une approche utilisée consiste à créer un automate de Büchi généralisé puis à appliquer une procédure appelée dégénéralisation (qui sera décrite section 5.7) pour obtenir un automate de Büchi avec un nombre d'états linéaire en le nombre de couleurs et d'états de l'automate de Büchi généralisé de départ.

Il a été montré [31] que dans le cadre de la traduction de formule LTL en automate de Büchi, utiliser une condition de Büchi généralisé portant sur les couleurs des transitions permet un gain significatif sur la taille de l'automate de Büchi résultant.

Pour comprendre l'idée derrière ces automates, prenons comme exemple "Je veux que b soit vrai infiniment souvent" (qui correspond à la formule LTL GF(b)). Avec une condition portant sur les états, il faut un état acceptant décrivant que l'on vient de voir un b et un état non-acceptant dans lequel on va rester si b ne sera plus vu. Cet automate est décrit dans la figure 3.4a. À l'inverse associer la couleur aux transitions comme dans la figure 3.4b permet de ne plus avoir besoin que d'un état et la condition est alors de voir infiniment souvent (l'arête portant) b.

Remarque 16. *Tout automate pour lequel la condition porte sur la coloration des états peut adopter une condition portant sur la coloration des transitions. Pour cela toute couleur présente sur un état est poussée sur les transitions sortantes de cet état.* [4]

3.3.4. TELA

Nous allons maintenant étendre l'utilisation d'arêtes colorées aux conditions d'Emerson-Lei. Ces automates sont nommés TELA pour Transition-based Emerson-Lei Automata et ils seront, sauf mention contraire, le type d'ω-automates utilisé dans le reste de cette thèse.

De manière formelle, ces automates sont définis ainsi :

Définition 33 (Automate d'Emerson-Lei basé sur les transitions, TELA)**.** *Un automate d'Emerson-Lei basé sur les transitions (Transition-based Emerson-Lei automaton ou TELA) est un tuple* $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ *où :*

4. Dans Spot les couleurs sont portées par les transitions mais cette méthode permet la gestion de la coloration des états. Voir `https://spot.lrde.epita.fr/concepts.html#trans-acc`

— Q *est un ensemble fini d'états ;*
— M *est un ensemble fini de couleurs ;*
— Σ *est un alphabet d'entrée fini ;*
— $\delta \subseteq Q \times \Sigma \times 2^M \times (2^Q \setminus \varnothing)$ *est un ensemble fini de transitions ;*
— $q_0 \in Q$ *est l'état initial ;*
— $\alpha \in \mathcal{C}(M)$ *est une condition d'Emerson-Lei portant sur les éléments de* M.

Notation 5. *Pour tout* $(q_1, \ell, A, q_2) \in Q \times \Sigma \times 2^M \times Q$, *on note* $d = q_1 \xrightarrow{\ell, A} q_2$ *s'il existe* $Q' \subseteq Q$ *tel que* $q_2 \in Q'$ *et* $(q_1, \ell, A, Q') \in \delta$.

Définition 34 (Exécution d'un ω-automate)**.** *Une exécution* r *sur* $\mathcal{A}$ *associée à un mot* $(\ell_i)_{i\in\mathbb{N}}$ *est une suite infinie de transitions* $r = (s_i \xrightarrow{\ell_i, A_i} s'_i)_{i\geq 0} \in \delta^\omega$ *telle que* $s_0 = q_0$ *et* $\forall i \geq 0, s'_i = s_{i+1}$.

Remarque 17. *Puisque* Q *est fini et l'ensemble des transitions est fini, alors lors d'une exécution infinie* r, *il doit exister une position* $j_r > 0$ *telle que pour tout* $i \geq j_r$ *la transition* $s_i \xrightarrow{\ell_i, A_i} s'_i$ *apparaît infiniment souvent dans* r.

Définition 35 (Automate déterministe)**.** *Un automate est déterministe si* $\forall q \in Q, \forall \ell \in \Sigma, \{(a, b, c, d) \in \delta \mid a = q, b = \ell\} \leq 1$.

Définition 36 (Automate complet)**.** *Un automate est complet si* $\forall q \in Q, \forall \ell \in \Sigma, \{(a, b, c, d) \in \delta \mid a = q, b = \ell\} \geq 1$.

Définition 37 (Exécution acceptante)**.** *Soit* $\mathsf{Rep}(r) = \bigcup_{i\geq j_r} A_i$ *l'ensemble des couleurs répétées infiniment souvent dans* r.

Une exécution r *est acceptante si* $\mathsf{Rep}(r) \vDash \alpha$ *et* $\mathcal{A}$ *accepte le mot* $(\ell_i)_{i\geq 0} \in \Sigma^\omega$. *Sinon elle est dite rejetante.*

Définition 38. *Le langage reconnu par un TELA* $\mathcal{A}$ *donné est l'ensemble des mots acceptés par* $\mathcal{A}$ *et est noté* $\mathcal{L}(\mathcal{A})$.

Définition 39 (Complément d'un langage)**.** *Étant donné un langage* L, *on note son complément* $L^C = \Sigma^\omega \setminus L$.

Notation 6. *Soit* $e = (q, a, m, q') \in \delta$ *une arête. On note* $\Gamma(e) = m$ *ses couleurs ; par extension, pour un ensemble* $\delta' \subseteq \delta$, *on note* $\Gamma(\delta') = \bigcup_{e\in\delta'} \Gamma(e)$ *l'union des couleurs portées par ces arêtes.*

Puisqu'un TELA $\mathcal{A}$ repose sur un graphe que nous désignerons par $G_\mathcal{A}$, nous pouvons y définir des notions portant sur ce graphe sous-jacent :

Définition 40 (Cycle d'un TELA)**.** *Un cycle* ℓ *d'un TELA* $\mathcal{A}$ *est un cycle du graphe sous-jacent. Un cycle est acceptant (resp. rejetant) si* $\Gamma(\ell) \vDash \alpha$ *(resp.* $\Gamma(\ell) \nvDash \alpha$*). L'ensemble des états d'un cycle* ℓ *est* $States(\ell) = \{q \in Q \mid$ *il existe* $e \in \ell$ *passant par* $q\}$. *L'ensemble des cycles de* $\mathcal{A}$ *est noté* $\mathcal{C}(\mathcal{A})$. *Cet ensemble est ordonné par la relation d'inclusion.*

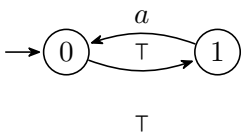


FIGURE 3.5. – Automate reconnaissant tous les mots ayant a à une position impaire

Définition 41 (Complément d'un TELA)**.** *Soit* $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ *un TELA déterministe complet. On définit* $\mathit{Complement}(\mathcal{A})$ *comme l'automate* $(Q, M, \Sigma, \delta, q_0, \neg\alpha)$ *(voir définition 31). On a alors* $\mathcal{L}(\mathit{Complement}(\mathcal{A})) = \mathcal{L}(\mathcal{A})^C$.

Dans la suite de ce manuscrit, nous utiliserons les abréviations suivantes :

— *DPA* désignera un automate de parité déterministe ;

— *DELA* désignera un automate d'Emerson-Lei déterministe ;

— *NBA* désignera un automate de Büchi non-déterministe.

3.4. Lien entre logique LTL et ω-automate

En 1986, VARDI et WOLPER proposèrent une méthode de traduction de formule LTL en automate de Büchi non-déterministe.

Notons qu'un exemple de spécification ne pouvant pas être transformée en automate de Büchi déterministe est $\mathsf{FG}(a)$ (utilisé dans l'idée de la preuve du théorème 1).

Cependant les automates de Büchi non-déterministes peuvent être transformés en automates déterministes à condition quelconque et on en conclut donc que toute formule LTL peut être transformée en TELA.

À l'inverse, il n'est pas possible de transformer tout automate en formule LTL. Considérons l'automate de la figure 3.5. Le langage associé est "*À toute position impaire, a est vrai*".

Même si ce type d'automate ne sera pas utilisé ici, on peut cependant mettre en avant les automates alternants linéaires dont le pouvoir d'expressivité est le même que celui des formules LTL. La traduction d'un tel automate en formule LTL existe [49] tout comme l'opération inverse [30].

Hiérarchie de Manna et Pnueli

En 1990, MANNA et PNUELI donnèrent une classification de langages sur les mots infinis. CERNA et PELÁNEK associèrent à chacune de ces classes des conditions d'acceptation ainsi que des restrictions sur la fonction de transition d'automates qui leurs sont associés.

Les différentes classes de langage qui sont introduites sont :

— La classe de *sûreté* décrivant qu'un évènement doit toujours arriver. Par exemple, le feu tricolore doit toujours afficher une seule couleur ;

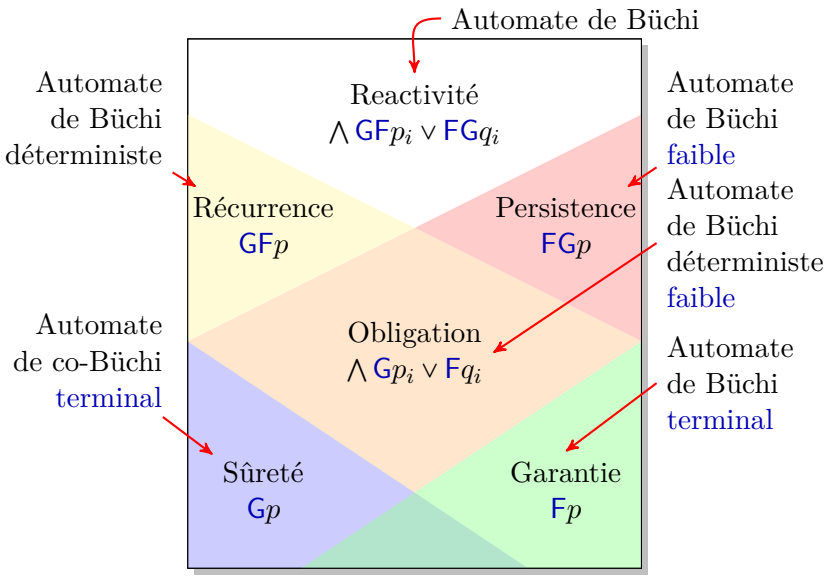


FIGURE 3.6. – Hiérarchie temporelle de Manna et Pnueli et les classes d'automates qui leur sont associés où p, p_i, q et q_i sont des formules ne comportant que des opérateurs Booléens, X et des opérateurs du passé [53]

— La classe de *garantie* décrivant qu'un évènement va arriver au moins une fois, comme l'existence d'au moins un moment où le feu va passer au vert ;

— La classe de *récurrence* décrivant un évènement qui doit se produire infiniment souvent comme le fait de devoir passer infiniment souvent du vert au rouge ;

— La classe de *persistence* comporte les formules qui doivent toujours être vérifiées à partir d'un certain moment ;

— Les formules d'*obligation* sont des combinaisons Booléennes de formules de sûreté et de garantie ;

— Les formules de *réactivité* sont des combinaisons Booléennes de formules de récurrence et de persistence.

Considérons une formule LTL de récurrence. Alors elle fait partie de la classe des formules de réactivité et peut donc être reconnue par un automate de Büchi non-déterministe. Cependant l'appartenance à la classe des récurrences implique l'existence d'un automate de Büchi déterministe reconnaissant la formule.

Remarque 18. *Une formule de persistence peut être vue comme le dual d'une condition de récurrence. Ainsi une formule de cette classe peut être reconnue par un automate de co-Büchi déterministe.*

À partir d'une formule de persistence ϕ, on peut construire un automate de Büchi pour $\neg\phi$ et complémenter cet automate pour obtenir un automate de co-Büchi associé à ϕ.

Le même raisonnement existe pour les formules de sûreté et de garantie.

Il existe des algorithmes ne pouvant être appliqués que sur certaines de ces classes de formules. Par exemple nous verrons qu'il est possible d'obtenir un automate minimal associé à une formule d'obligation. Pour décider si une formule fait bien partie d'une telle classe, nous allons nous baser sur une analyse syntaxique de celle-ci. Pour cela, introduisons la grammaire suivante :

$$
\begin{aligned}
\varphi_B &::= \top \mid \bot \mid v \mid \neg\varphi_B \mid \varphi_B \wedge \varphi_B \mid (\varphi_B \vee \varphi_B) \mid \varphi_B \leftrightarrow \varphi_B \mid \varphi_B \oplus \varphi_B \mid \varphi_B \rightarrow \varphi_B \mid \mathsf{X}\varphi_B \\
\varphi_G &::= \varphi_B \mid \neg\varphi_S \mid \varphi_G \wedge \varphi_G \mid (\varphi_G \vee \varphi_G) \mid \varphi_S \rightarrow \varphi_G \mid \mathsf{X}\varphi_G \mid \mathsf{F}\varphi_G \mid \varphi_G\mathsf{U}\varphi_G \mid \varphi_G\mathsf{M}\varphi_G \\
\varphi_S &::= \varphi_B \mid \neg\varphi_G \mid \varphi_S \wedge \varphi_S \mid (\varphi_S \vee \varphi_S) \mid \varphi_G \rightarrow \varphi_S \mid \mathsf{X}\varphi_S \mid \mathsf{G}\varphi_S \mid \varphi_S\mathsf{R}\varphi_S \mid \varphi_S\mathsf{W}\varphi_S \\
\varphi_O &::= \varphi_G \mid \varphi_S \mid \neg\varphi_O \mid \varphi_O \wedge \varphi_O \mid (\varphi_O \vee \varphi_O) \mid \varphi_O \leftrightarrow \varphi_O \mid \varphi_O \oplus \varphi_O \mid \varphi_O \rightarrow \varphi_O \\
&\quad \mid \mathsf{X}\varphi_O \mid \varphi_O\mathsf{U}\varphi_G \mid \varphi_O\mathsf{R}\varphi_S \mid \varphi_S\mathsf{W}\varphi_O \mid \varphi_G\mathsf{M}\varphi_O \\
\varphi_P &::= \varphi_O \mid \neg\varphi_R \mid \varphi_P \wedge \varphi_P \mid (\varphi_P \vee \varphi_P) \mid \varphi_P \leftrightarrow \varphi_P \mid \varphi_P \oplus \varphi_P \mid \varphi_P \rightarrow \varphi_P \\
&\quad \mid \mathsf{X}\varphi_P \mid \mathsf{F}\varphi_P \mid \varphi_P\mathsf{U}\varphi_P \mid \varphi_P\mathsf{R}\varphi_S \mid \varphi_S\mathsf{W}\varphi_P \mid \varphi_P\mathsf{M}\varphi_P \\
\varphi_R &::= \varphi_O \mid \neg\varphi_P \mid \varphi_R \wedge \varphi_R \mid (\varphi_R \vee \varphi_R) \mid \varphi_R \leftrightarrow \varphi_R \mid \varphi_R \oplus \varphi_R \mid \varphi_R \rightarrow \varphi_R \\
&\quad \mid \mathsf{X}\varphi_R \mid \mathsf{G}\varphi_R \mid \varphi_R\mathsf{U}\varphi_G \mid \varphi_R\mathsf{R}\varphi_R \mid \varphi_R\mathsf{W}\varphi_R \mid \varphi_G\mathsf{M}\varphi_R
\end{aligned}
$$

Dans cette grammaire φ_X correspond à la structure de formules de la classe X où

— B correspond aux formules qui sont à la fois des formules de sûreté syntaxique et de garantie syntaxique ;

— G correspond aux formules de *garantie syntaxique* ;

— S correspond aux formules de *sûreté syntaxique* ;

— O correspond aux formules d'*obligation syntaxique* ;

— P correspond aux formules de *persistence syntaxique* ;

— R correspond aux formules de *récurrence syntaxique*.

Remarque 19. *Pour ces classes, le fait de ne pas appartenir à la grammaire ne signifie pas ne pas appartenir à la classe. Par exemple la formule* $\mathsf{GF}(a) \vee \mathsf{FG}(a)$ *ne correspond à aucune règle de* φ_R*. Cependant elle est équivalente à* $\mathsf{GF}(a)$ *qui est bien une formule de récurrence.*

Proposition 1. *À toute formule d'une classe* X *on peut associer une formule de la classe syntaxique associée équivalente.*

De plus, si une formule correspond à une classe syntaxique, il n'est pas garanti qu'elle n'appartienne pas à une de ses sous-classes.

Par exemple la formule $\mathsf{FG}(a \vee \bar{a})$ *est une persistence syntaxique mais pas une garantie syntaxique. Cependant elle est équivalente à* $\top$ *et est donc une garantie.*

Dans Spot, des fonctions permettent de détecter si une formule peut être associée à une de ces règles. Il s'agit des fonctions de la forme `is_syntactic_X()` où `X` est le nom de la classe de formule [5].

5. Voir `https://spot.lrde.epita.fr/doxygen/classspot_1_1formula.html#adef55f28ad9a89e9bce6f6690a70367d` pour la documentation de ces fonctions

3.5. Jeux

Nous allons maintenant évoquer les jeux. Ils peuvent être étudiés au travers de la théorie des jeux. Basée entre autres sur les travaux de Ernst Zermelo [75], cette théorie trouve des applications dans le domaine des sciences sociales par exemple [71] mais aussi du sport [73] et évidemment les jeux tels que le Monopoly ou les échecs.

3.5.1. Caractéristiques

Nous allons ici nous intéresser à des jeux dont les propriétés rappellent celles des échecs. Ils sont caractérisés par plusieurs points.

Le premier concerne les informations connues par chaque joueur. Un joueur connaît :

— toutes les actions qu'il peut faire ;

— toutes les actions que peut faire son adversaire ;

— l'impact de ses actions par rapport à la condition de victoire.

Par exemple le poker n'est pas concerné car on ne connaît pas les cartes des adversaires.

Une autre limitation aux jeux que nous allons étudier se situe dans l'absence de hasard.

Même si nous n'allons travailler qu'avec des jeux dits séquentiels où il y a une alternance entre les joueurs (à l'inverse de pierre-papier-ciseaux) nos définitions ne prendront pas toujours en compte cette restriction.

Enfin, dans ces jeux les parties seront infinies et la condition de victoire sera adaptée à cette absence de fin.

Puisque seuls ces types de jeux seront étudiés, ils seront simplement désignés par "jeux".

Nous allons maintenant avoir besoin de définir une représentation de ces jeux.

Dans le cas des jeux finis une manière naturelle de représenter l'ensemble des parties possibles est un arbre où chaque nœud porte un état de la partie et où l'ensemble des fils d'un nœud s est l'ensemble des états qui peuvent être atteint en un seul coup depuis l'état associé à s. Par exemple dans la figure 3.7, la racine porte le plateau vide et pour passer au niveau suivant, il faut placer une croix n'importe où sur le plateau. Le passage au dernier niveau représenté se fait alors en plaçant un cercle sur n'importe quelle case vide.

Remarque 20. *Dans la figure 3.7, à chaque niveau (et donc chaque nœud), on peut associer le joueur qui doit faire une action. Il y a ainsi une partition des nœuds de cet arbre dépendant du joueur qui va modifier l'état du jeu.*

Cependant nous travaillons sur des jeux infinis. Considérons alors un autre jeu où les deux joueurs donnent un entier étant 0 ou 1 mais pas de manière simultanée. Pour gagner le second joueur ne doit jamais donner le même numéro que son adversaire. Vu que le joueur cherche à gagner, il va toujours donner un nombre différent de celui de son adversaire (car il le connaît).

Cela nous amène à un arbre (infini) représentant les parties de ce jeux donné dans la figure 3.8a.

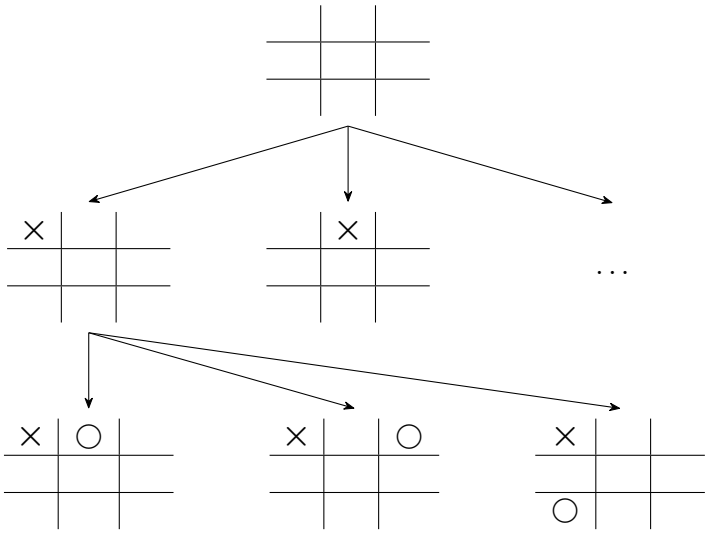


FIGURE 3.7. – Représentation partielle de l'ensemble des parties possibles pour le jeu du morpion

Nous considérons que si le second joueur a perdu (état rouge) ils continuent à jouer. Il y a cinq états possibles pour le jeu, formés par la combinaison des critères suivants :

— le joueur qui s'apprête à jouer (états carrés pour le premier) ;

— si le second joueur a déjà perdu (états masqués accessibles depuis les états rouges) ;

— pour le second joueur, s'il n'a pas perdu et doit jouer 0 (vert) ou 1 (marron) pour ne pas perdre tout de suite.

Dans cet arbre infini, il est possible de fusionner les états équivalents (de même couleur), c'est-à-dire les états décrivant le même état de la partie. On obtient alors le jeu de la figure 3.8b qui est une représentation de ce jeu proche de celle des automates. Nous conservons l'idée des états mais maintenant à chaque état est associé un joueur. Pour chaque état, les arêtes qui en sortent portent les actions que peut faire le joueur qui lui est associé.

Définition 42 (Jeu)**.** *Un jeu est un TELA de la forme* $\mathcal{G} = (Q, M, \Sigma, \delta, q_0, \alpha)$ *tel que :*

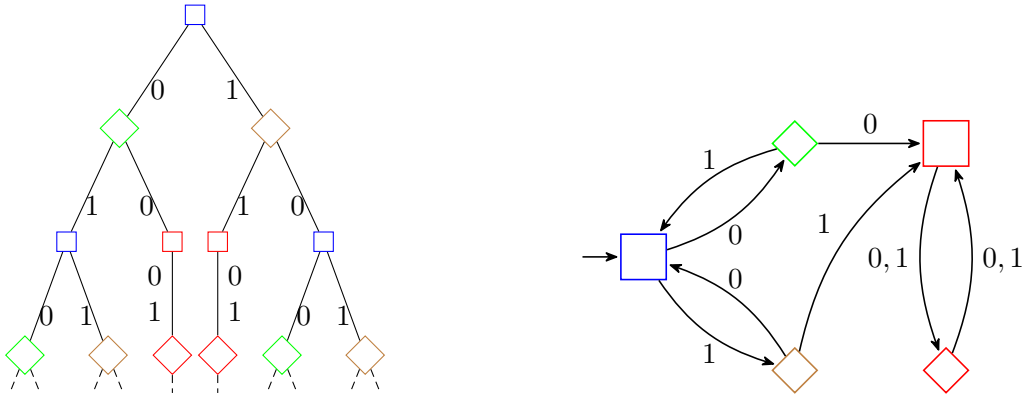
— $Q_0 \cup Q_1$ *où* Q_0 *et* Q_1 *sont deux ensembles disjoints d'états respectivement contrôlés par le joueur 0 et le joueur 1 ;*

— $\delta : Q \times \Sigma \to 2^M \times Q$ *est un ensemble fini de transitions ;*

— q_0 *est l'état initial ;*

— α *est une condition d'Emerson-Lei portant sur les éléments de* M *;*

Définition 43 (Jeu de parité)**.** *Un jeu de parité est un jeu pour lequel la condition* α *est une condition de parité.*



(a) Arbre partiel des parties de notre jeu

(b) Automate associé à l'arbre de la figure 3.8a obtenu en fusionnant les états équivalents

FIGURE 3.8. – Représentation de l'ensemble des parties possibles de notre jeu sous forme d'arbre et d'automate.

Définition 44 (Partie)**.** *Une exécution d'un jeu $\mathcal{G}$ est appelée partie. Elle est dite gagnante pour le joueur 1 si l'exécution est acceptante et perdante pour le joueur 1 sinon.*

Ici la définition de jeu n'implique pas une alternance des joueurs. L'ajout de cette alternance donne ce que l'on nomme des jeux de Gale-Stewart.

Puisqu'il s'agit des jeux que nous allons utiliser, nous allons les désigner par jeu même si certaines définitions que nous donnerons resteront valables pour la définition précédente de jeu.

Définition 45 (Jeu de Gale-Stewart)**.** *Un jeu de Gale-Stewart est un TELA de la forme $\mathcal{G} = (Q_0 \cup Q_1, M, \Sigma, \delta, q_0, \alpha)$ tel que :*

- *Q_0 et Q_1 sont deux ensembles disjoints d'états respectivement contrôlés par le joueur 0 et le joueur 1 ;*
- *$\delta \subseteq (Q_0 \times \Sigma \times 2^M \times Q_1) \cup (Q_1 \times \Sigma \times 2^M \times Q_0)$ est un ensemble fini de transitions ;*
- *q_0 est l'état initial ;*
- *α est une condition d'Emerson-Lei portant sur les éléments de M.*

3.5.2. Stratégie

Un autre point important de la théorie des jeux qui nous sera utile est ce que l'on appelle "stratégie". De manière générale, une stratégie est un moyen pour un joueur de choisir tout au long d'une partie les actions qu'il effectue. Ce choix pouvant aussi s'appuyer sur l'impact des actions qu'il a effectué plus tôt dans la partie. Ici nous n'allons considérer que des jeux pour lesquels la stratégie ne dépend que de l'état de la partie. De manière formelle, cette stratégie sans mémoire est définie de la manière suivante :

Définition 46 (Stratégie sans mémoire)**.** *Une stratégie sans mémoire pour le joueur* $i \in \{0,1\}$ *est une fonction* $\sigma : Q_i \to \delta$ *telle que* $\sigma(q)$ *est toujours une transition sortante de* $q \in Q_i$*.*

Définition 47 (Partie consistante avec une stratégie)**.** *Une partie* $p = (s_j \xrightarrow{\ell_j, A_j} s'_j)_{j \geq 0}$ *est dite consistante avec une stratégie* σ *pour un joueur* i *si* $\forall s_j \in Q_i, \sigma(s_j) = (s_j \xrightarrow{\ell_j, A_j} s'_j)$*, c'est-à-dire si chaque action du joueur* i *a été déterminée par* σ*.*

Définition 48 (Stratégie gagnante)**.** *Une stratégie* σ *est dite gagnante pour le joueur 1 (respectivement 0) depuis un ensemble* W *d'états si toute partie commençant depuis un état de* W *et consistante avec* σ *est gagnante (respectivement perdante) pour le joueur 1.*

Un tel ensemble W *est qualifié de région gagnante.*

Théorème 2 (Existence de stratégie dans un jeu de parité [54])**.** *Il existe une unique partition* (W_0, W_1) *de* $Q_0 \cup Q_1$ *telle qu'il existe une stratégie sans mémoire* σ_0 *(respectivement* σ_1*) gagnante pour le joueur* 0 *(respectivement* 1*) depuis* W_0 *(respectivement* W_1*).*

De manière générale, trouver une stratégie gagnante pour un des joueurs est complexe. Dans le cas des jeux portant une condition de Rabin (respectivement Streett), le problème est NP-complet (respectivement co-NP-complet) [23] et Piterman et al. ont décrit un moyen de résoudre de tels jeux avec une complexité en temps et en espace en $O(n^{k+1}k!)$ où n est le nombre d'états et k le nombre de paires de Rabin ou de Streett dans la condition d'acceptation du jeu [60].

Dans des jeux de Rabin ou Streett, seul l'un des deux joueurs est assuré de posséder une stratégie sans mémoire depuis sa région gagnante. L'autre joueur possède une stratégie qui nécessite de retenir l'historique des états visités dans la partie. Les jeux de parité, qui sont à la fois des jeux de Rabin et de Streett, sont un cas particulier où les stratégies gagnantes sont sans mémoire pour les deux joueurs [32].

Pour ces jeux de parité, extraire une telle stratégie a une complexité en temps de $O(n^{O(\log(k))})$ où n est le nombre d'états du jeu et k le nombre de couleurs [11].

Une comparaison de différentes méthodes de résolution de tels jeux a déjà été effectuée [29] et a conclu que l'algorithme récursif de Zielonka est le plus rapide en pratique.

Puisqu'il s'agit du seul type de stratégie que nous étudierons, nous désignerons les stratégies sans mémoire par stratégie.

3.6. Synthèse LTL

Toutes les notions qui ont été données nous servirons à décrire comment nous résolvons de problème de la synthèse LTL.

Décrit en 1989 par Pnueli et Rosner sous le nom d'*implementability problem*, le problème de la synthèse LTL consiste à prendre une spécification (formule LTL) décrivant un comportement qu'un système réactif (contrôleur) doit adopter. Celui-ci associe à

un flux de propositions atomiques d'entrées une suite d'assignations de propositions atomiques de sortie.

De manière plus formelle, ce problème est défini de la manière suivante :

Définition 49 (Synthèse LTL, [62]). *Le problème de la synthèse LTL est le problème prenant en entrée un ensemble $I = \{i_1, \ldots, i_n\}$ de propositions atomiques d'entrées, un ensemble $O = \{o_1, \ldots, o_m\}$ de propositions atomiques de sorties ainsi qu'une formule LTL L et qui consiste à donner une fonction f prenant en entrée une suite d'assignations des propositions de I et produisant une suite d'affectations des propositions de O respectant la spécification L.*

Définition 50 (Contrôleur). *La fonction f de la définition 49 sera désignée par contrôleur.*

Faisons un parallèle avec l'exemple du feu de circulation introduit dans le deuxième chapitre. Nous avions demandé entre autres à ce que la présence d'une voiture devant un feu implique que le feu associé finira par passer au vert.

Mettons en avant plusieurs caractéristiques de ce problème.

La première d'entre elles est qu'il peut être impossible de créer un système pour certaines spécifications. Demandons par exemple à ce que le premier capteur détecte toujours une voiture ($\mathsf{G}(C_1)$). Pour satisfaire une telle contrainte, il faudrait qu'il y ait toujours une voiture devant le premier feu alors que notre système ne contrôle que la couleur des feux et pas le flux de véhicules. Un autre exemple serait de demander à ce qu'une ampoule soit allumée et éteinte en même temps.

Déterminer s'il est possible de créer un système respectant des spécifications peut être vu comme un sous-problème de celui de la synthèse LTL. Ce problème est nommé problème de réalisabilité.

Définition 51 (Problème de réalisabilité). *Le problème de réalisabilité LTL est un problème prenant en entrée un ensemble $I = \{i_1, \ldots, i_n\}$ de propositions atomiques d'entrées, un ensemble $O = \{o_1, \ldots, o_m\}$ de propositions atomiques de sorties ainsi qu'une formule LTL L et qui consiste à déterminer s'il existe (au moins) une fonction f prenant en entrée une suite d'assignations de propositions de I et produisant une suite d'assignations des propositions de O respectant la spécification L.*

Une deuxième caractéristique du problème de la synthèse LTL est qu'il n'admet pas forcément une unique solution. Nous indiquions par exemple page 13 que lorsque personne n'attend, l'ampoule verte peut être allumée ou non.

Même s'il ne s'agit pas d'une caractéristique du problème, une notion de qualité de la solution peut être ajoutée. Ainsi pour notre feu de circulation, le fabriquant peut chercher à créer un circuit électrique le moins cher possible alors que le client peut vouloir limiter la taille du circuit de manière à réduire le délai entre l'arrivée des signaux d'entrée et la production des signaux de sortie [6] ou encore pour produire un circuit physiquement plus petit.

6. Ce délai de propagation de signal dépend en réalité davantage du plus long chemin entre une entrée et une sortie pondérée par le type des portes le long de ce chemin.

Le problème de la synthèse LTL est dans la classe des problèmes 2EXPTIME-complets [45], ce qui limite en pratique pour le moment son utilisation.

4. Synthèse LTL

Le but de ce chapitre est de donner une idée de la manière dont est traité le problème de la synthèse LTL par divers outils. Il sera surtout l'occasion de donner une vue détaillée de la manière dont `ltlsynt`, l'outil de synthèse distribué avec Spot, procède pour résoudre ce problème. Cette description fait l'objet d'un article soumis pour une édition spéciale du journal FMSD [66]. Cela permettra d'introduire et de localiser les diverses contributions qui seront décrites dans les chapitres suivants.

4.1. Définitions

Avant de décrire comment est traité le problème de la synthèse LTL, il nous faut introduire quelques définitions.

Nous avons déjà décrit le problème de la synthèse LTL comme la recherche d'un contrôleur, s'il peut exister, à partir d'une spécification ainsi que celui de réalisabilité dont le but est de savoir si, étant donnée une spécification, un tel contrôleur peut exister.

Ces deux problèmes sont ceux qui sont proposés lors de la SYNTCOMP [40]. Dans ce concours visant à comparer des outils de synthèse, le contrôleur doit être sous forme d'*And-Inverter Graph* (AIG).

Il s'agit d'un graphe acyclique représentant un circuit logique synchrone, c'est-à-dire un circuit logique avec une notion de temps discret. Ces circuits ne peuvent être composés que de :

— portes *ET* retournant 1 si et seulement si leurs deux entrées valent 1 ;

— portes *NON* retournant 1 si et seulement si son entrée vaut 0 ;

— *bascules* renvoyant 0 initialement puis la valeur de son entrée à l'instant précédent.

Exemple 8. *Le circuit décrit dans la figure 4.1 représente un circuit AIG. Il y est représenté comme avec Spot : les cercles correspondent aux portes ET alors que les arêtes portant un point indiquent les négations. Les bascules sont divisées en deux parties. La première de la forme latch_in ne comporte qu'une entrée et est liée à latch_out qui renvoie la valeur de latch_in à l'instant précédent (0 au début).*

Ce circuit associe donc à la variable de sortie 0 comme première valeur puis 1 si et seulement si au moment précédent, i_0 *valait 0 et* i_1 *valait 1.*

Pour encoder de tels graphes, un format nommé *AIGER*[1] a été introduit en 2007 [8] et c'est sous cette forme que les outils participant à la SYNTCOMP vont fournir le contrôleur.

1. Référence à l'Eiger en Suisse où a eu lieu l'Alpine Verification Meeting 2006.

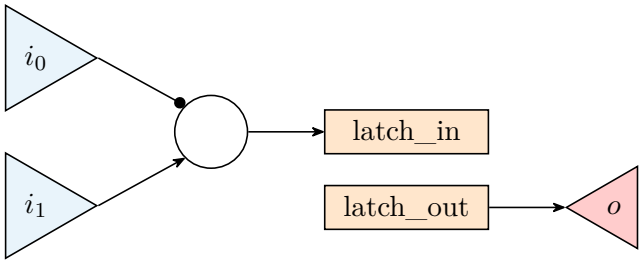


FIGURE 4.1. – Circuit associant à o la valeur 0 dans un premier temps puis 1 si et seulement si à l'instant précédent, i_0 valait 0 et i_1 valait 1

Avec Spot, il est possible de manipuler les circuits AIG. Par exemple un circuit peut être lu lorsqu'il est donné sous une forme restreinte d'AIGER. Il est également possible d'encoder un circuit AIG en AIGER.

4.2. ltlsynt

Abordons maintenant l'outil `ltlsynt` de Spot pour lequel l'ensemble du travail décrit a été implémenté.

4.2.1. Processus global

Commençons par une vue globale du processus avant de décrire rapidement les différences impliquées par la valeur de l'option `--algo`.

Omettons le découpage de formule et l'analyse de la structure de la formule. La première étape décrite dans la figure 4.2 consiste alors à traduire la spécification en un jeu de parité. Nous verrons qu'il s'agit souvent de deux étapes. La première consiste à traduire la formule LTL en un automate de parité (figure 4.3).

Puisque nous cherchons à manipuler les jeux (figure 4.10), il nous faut alors séparer les variables contrôlables de celles qui ne le sont pas. Le joueur 0 (environnement) choisit les valuations des variables d'entrée alors que le joueur 1 (contrôleur) choisit les valuations des variables de sortie. Ce jeu est :

— déterministe : pour chaque joueur, chaque action ne peut mener qu'à un état ;

— complet sur les entrées : depuis chaque état du joueur 0, toute valuation des variables d'entrée doit mener à un état.

Dans ce jeu, s'il est possible pour le joueur 1 de trouver une stratégie gagnante depuis l'état initial, alors il sera possible de synthétiser une solution. En effet, une partie y est acceptante pour le joueur 1 si la suite d'entrées sorties respecte la spécification. Extraire une stratégie gagnante pour le joueur 1 signifie donc que pour toute suite d'entrées, il est possible de fournir une suite d'assignations des variables de sorties respectant la spécification. Cette stratégie est encodée sous forme de machine de Mealy incomplètement spécifiée (figure 4.5).

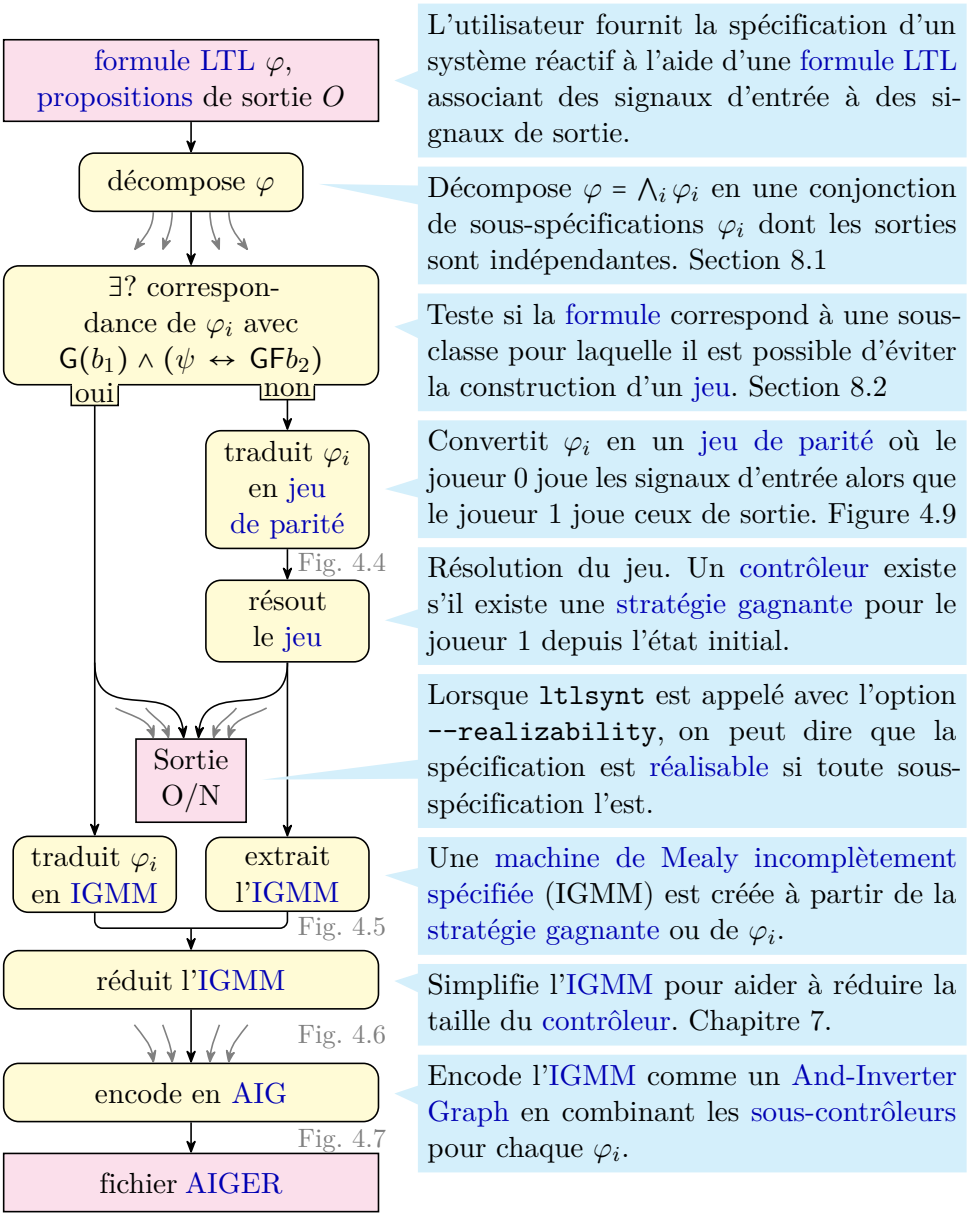


FIGURE 4.2. – Processus de résolution du problème de synthèse LTL par `ltlsynt`

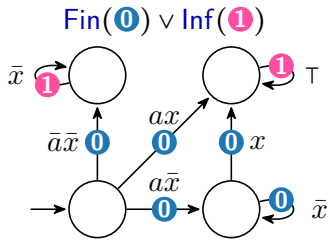


FIGURE 4.3. – Automate de parité maximale impaire pour $a \leftrightarrow \mathsf{F}(x)$

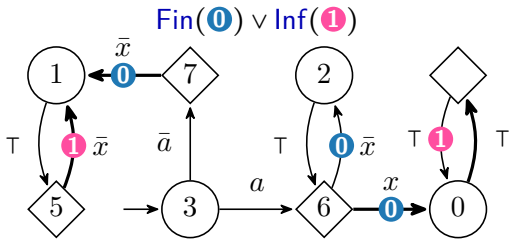


FIGURE 4.4. – Jeu de parité obtenu à partir de l'automate de la figure 4.3. Le joueur 0 joue depuis les nœuds ronds et choisit les signaux d'entrée alors que le joueur 1 joue depuis les états carrés et joue les signaux de sortie.

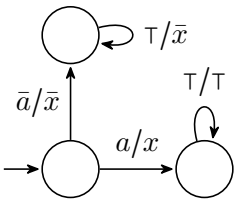


FIGURE 4.5. – Stratégie de la figure 4.4 vue comme une machine de Mealy incomplètement spécifiée

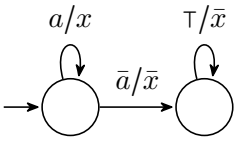


FIGURE 4.6. – Machine de Mealy réduite pour la stratégie gagnante de la figure 4.4. Les états de cette machine sont utilisés pour savoir si $\bar{a}$ a été vu.

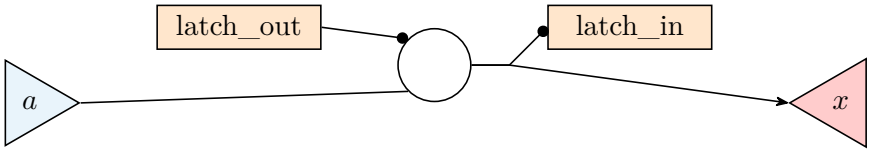


FIGURE 4.7. – And-Inverter Graph associé à la machine de Mealy de la figure 4.6. La sortie x est vraie aussi longtemps que a reste vraie et devient faux dès la première occurrence de $\bar{a}$.

La liberté offerte par la stratégie permet ensuite de réduire sa taille (figure 4.6, chapitre 7).

Ce processus global est précédé d'un découpage du problème de manière à créer une solution globale à partir de contrôleurs liés à des sous-formules de la spécification de départ. Cette fusion des contrôleurs se fait au moment d'encoder la solution sous forme de circuit AIG.

Remarque 21. *Un travail important autour de la création de l'AIG à partir de la stratégie (sous forme de machine de Mealy incomplètement spécifiée) a été effectué par Philipp Schlehuber-Caissier en parallèle du travail présenté dans ce manuscrit [66].*

Pour certaines formules, `ltlsynt` est également capable de se passer de la création d'un jeu de parité. Cette contribution qui sera décrite dans la section 8.2 permet pour ces formules de détecter si la spécification est réalisable sans construire d'automate. Si une telle solution existe, elle peut être fournie en ne traduisant qu'une partie de la formule.

4.2.2. Création du jeu de parité à partir d'une formule LTL

Nous allons ici décrire les différentes possibilités offertes par `ltlsynt` pour créer un jeu de parité à partir d'une formule LTL. Pour cela nous commencerons par évoquer diverses méthodes pour obtenir un DPA à partir d'une formule LTL puis nous verrons comment certaines d'entre elles sont utilisées dans `ltlsynt` pour obtenir un jeu de parité. Nous verrons également la méthode utilisée pour transformer une formule en un automate à condition quelconque.

Commençons par la construction d'un jeu de parité à partir d'une formule LTL.

Cette étape peut généralement être découpée en deux parties : la première consiste à traduire une formule LTL en automate de parité alors que la seconde consiste à transformer cet automate en jeu.

Il existe plusieurs manières d'effectuer la première transformation et une liste non-exhaustive est présentée dans la figure 4.8. Ces méthodes s'appuient sur la construction d'automates intermédiaires. Ainsi il est par exemple possible de créer un automate de Büchi généralisé non-déterministe à partir d'une formule LTL, d'appliquer une dégénéralisation pour obtenir un automate de Büchi non-déterministe. Il est transformé en un automate de Rabin déterministe sur lequel des algorithmes tels que IAR peuvent être appliqués pour obtenir un automate de parité.

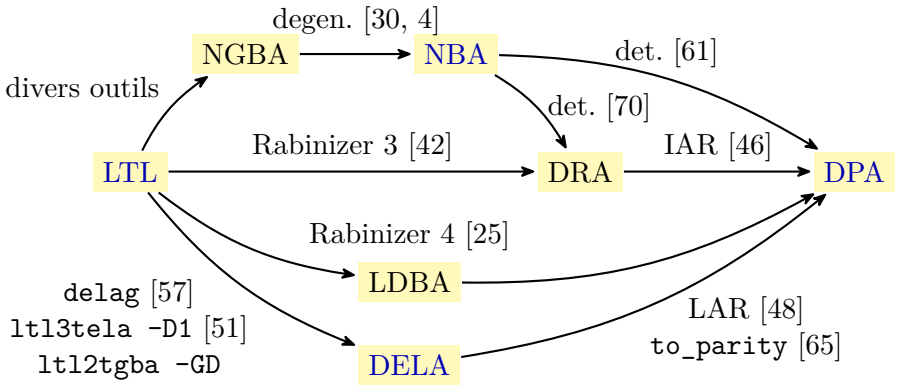


FIGURE 4.8. – Différentes méthodes d'obtention d'un automate de parité à partir d'une formule LTL. En plus des classes d'automates habituelles, NGBA y désigne un automate de Büchi généralisé non-déterministe et LDBA signifie limit-deterministic Büchi automaton.

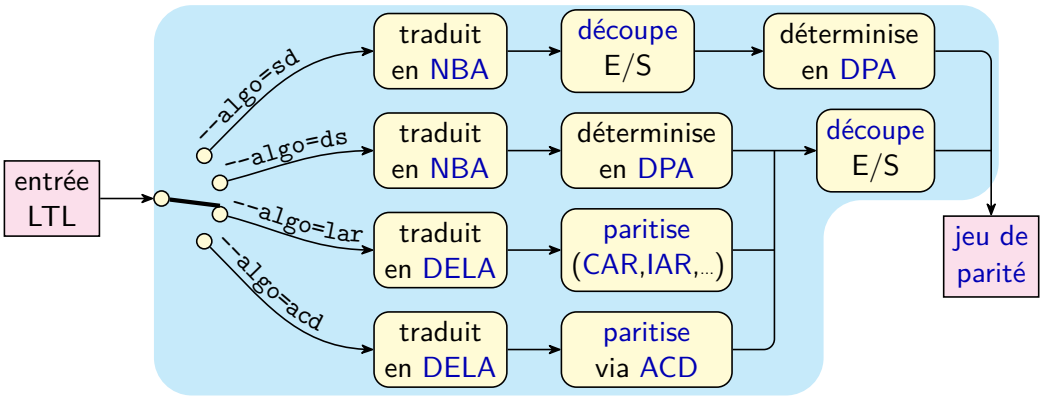


FIGURE 4.9. – Création d'un jeu de parité à partir d'une formule LTL avec `ltlsynt`

Décrivons les méthodes d'obtention d'un jeu de parité proposées par `ltlsynt`. Ces dernière sont liées à l'option `--algo` de `ltlsynt`.

Approche `ds` La première possibilité est nommée `ds` pour "determinize and split". Elle consiste à traduire dans un premier temps la formule LTL en un automate de Büchi non-déterministe à l'aide de la méthode standard de traduction de Spot [20].

Puisque nous travaillons avec des jeux déterministes, une procédure de déterminisation est ensuite appliquée à l'aide d'une variante de la déterminisation de Safra [64]. Cette méthode nous donne alors un automate de parité déterministe.

Enfin, les entrées et les sorties sont découpées de manière à obtenir un jeu.

Approche `sd` L'approche suivante nommée `sd` pour "split and determinize" effectue le même travail sauf que les étapes de découpage et de déterminisation sont inversées.

Cette inversion s'appuie sur la complexité de l'étape de déterminisation. Celle-ci dépend notamment de la taille de l'alphabet. Lorsque l'automate de Büchi non-déterministe est déjà découpé, à chaque instant de la déterminisation les seuls éléments de "l'alphabet" sont soit des entrées, soit des sorties.

Approches `lar` et `acd` Les approches `lar` et `acd` consistent toutes les deux à construire à partir de la formule LTL un automate déterministe à condition quelconque. Une procédure de paritisation permet ensuite de transformer ce TELA en automate déterministe de parité qui sera ensuite découpé de manière à obtenir un jeu de parité. La seule différence entre les deux procédures se situe dans la méthode de paritisation. L'approche `lar` s'appuie sur une procédure issue de la combinaison de multiples méthodes de transformation de condition de TELA [65] que nous décrirons dans le chapitre 5. La méthode `acd` s'appuie sur une méthode plus récente basée sur une analyse des cycles du TELA [14] qui sera décrite dans le chapitre 6.

Remarque 22. *L'option `--algo` peut également recevoir la valeur `lar.old`. Elle correspond à ce qu'était `lar` jusqu'en 2019 avant de devenir `lar.old` lors de l'introduction de la nouvelle procédure de paritisation qui sera décrite dans la section 5.1. Il s'agit de l'unique différence entre `lar` et `lar.old`.*

Transformation de l'automate de parité en jeu

Introduisons notre méthode de découpage des entrées et des sorties d'un automate de parité.

Pour rappel, dans le jeu que nous créons, le joueur 1 contrôle les variables de sortie alors que le joueur 0 est associé aux variables d'entrée et il y a une alternance entre les deux joueurs.

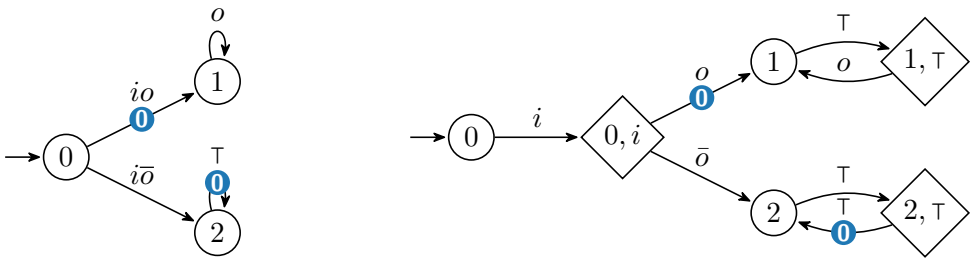
La procédure de découpage est définie pour tout type de condition d'acceptation et une partition en deux ensembles Q_0 et Q_1 de l'automate résultant permet d'associer à chaque état un joueur.

Définition 52 (Découpage d'un automate d'Emerson-Lei)**.** *Soit $\mathcal{A} = (Q, M, \mathbb{B}^{I \cup 0}, \delta, q_0, \alpha)$ un TELA. Le découpé de $\mathcal{A}$ est le TELA $\mathcal{A}_s = (Q_0 \uplus Q_1, M, \mathbb{B}^I \cup \mathbb{B}^O, \delta_0 \cup \delta_1, \alpha)$ où*

- $Q_0 = Q$;
- $Q_1 = Q \times \mathbb{B}^I$;
- $\delta_0 = \left\{ s \xrightarrow{\ell_i, \varnothing} (s, \ell_i) \,\middle|\, \ell_i \in \mathbb{B}^I, \ell_o \in \mathbb{B}^O, s \xrightarrow{\ell, A} d \in \delta, \ell = \ell_i \wedge \ell_o \right\}$;
- $\delta_1 = \left\{ (s, \ell_i) \xrightarrow{\ell_o, A} d \,\middle|\, \ell_i \in \mathbb{B}^I, \ell_o \in \mathbb{B}^O, s \xrightarrow{\ell, A} d \in \delta, \ell = \ell_i \wedge \ell_o \right\}$.

Définition 53 (Exécution sur un automate découpé)**.** *Soit $\mathcal{A}_s = (Q_0 \uplus Q_1, M, \mathbb{B}^I \cup \mathbb{B}^O, \delta_0 \cup \delta_1, \alpha)$ un TELA découpé, $r = (r_i)_{i \geq 0}$ une exécution et posons que pour tout i, $r_i = r_i^I r_i^0$ où $r_i^I \in \mathbb{B}^I$ et $r_i^O \in \mathbb{B}^O$ sont les projections de r_i.*

Le découpé de r est l'exécution $r_s = r_0^I r_0^O r_1^I r_1^0 \ldots \in (\mathbb{B}^I \cdot \mathbb{B}^O)^\omega$.



(a) Automate de Büchi

(b) Automate de Büchi de la figure 4.10a découpé

FIGURE 4.10. – Découpage d'un automate de Büchi avec $I = \{i\}$ et $O = \{o\}$. Les états ronds sont ceux du joueur 0 alors que le joueur 1 possède les états en forme de losange.

Remarque 23. *Un automate $\mathcal{A}$ accepte une exécution r si et seulement si $\mathcal{A}_s$ accepte r_s. Le découpage préserve donc la spécification sous-jacente.*

Exemple 9. *La figure 4.10 présente le découpage d'un automate de Büchi (figure 4.10a) en un automate découpé (figure 4.10b) où l'ensemble des entrées est $I = \{i\}$ alors que celui de sortie est $O = \{o\}$.*

Puisque depuis l'état 0 le seul élément de $\mathbb{B}^I$ qui est porté par une arête est $\{i\}$ alors on obtient dans l'automate résultant un état 0 qui est relié à un état $(0, \{i\})$ par une arête portant $\{i\}$. De même puisque depuis l'état 0, 1 est accessible en voyant o, on obtient une arête portant o de $(0, \{i\})$ à 1 dans l'automate découpé.

Méthode d'obtention d'un TELA à partir d'une formule LTL

Nous allons maintenant décrire comment est obtenu le TELA utilisé par `acd` et `lar`.

Pour cela nous aurons besoin de la définition de formule LTL sous forme normale négative :

Définition 54 (Formule sous forme normale négative)**.** *Une formule sous forme normale négative est une formule où la négation ne peut porter que sur des propositions atomiques et n'utilisant pas les opérateurs $\leftrightarrow$, $\rightarrow$ et $\oplus$. Elle suit donc la grammaire suivante :*

$$\varphi ::= \top \mid \bot \mid v \mid \neg v \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}(\varphi) \mid \varphi \mathsf{U} \varphi \mid \varphi \mathsf{R} \varphi \mid \mathsf{F}(\varphi) \mid \mathsf{G}(\varphi)$$

où v est une proposition atomique.

Remarque 24. *Toute formule LTL peut être réécrite sous forme normale négative.*

Nous utiliserons également un produit généralisé entre automates :

Théorème 3. *Soient $\mathcal{A}_1 = (Q_1, M_1, \Sigma, \delta_1, i_1, \alpha_1)$ et $\mathcal{A}_2 = (Q_2, M_2, \Sigma, \delta_2, i_2, \alpha_2)$ deux TELA déterministes sur le même alphabet utilisant deux ensembles de couleurs disjoints M_1 et M_2.*

Pour tout opérateur $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ on définit $\mathsf{Product}_\odot(\mathcal{A}_1, \mathcal{A}_2)$ comme le TELA $(Q, M, \Sigma, \delta, i, \alpha)$ où

— $Q = Q_1 \times Q_2$;
— $\delta = \{((s_1, s_2), \ell_1, m_1 \cup m_2, (d_1, d_2)) \mid (s_1, \ell_1, m_1, d_1) \in \delta_1, (s_2, \ell_2, m_2, d_2) \in \delta_2, \ell_1 = \ell_2\}$;
— $i = (i_1, i_2)$;
— $\alpha = \alpha_1 \odot \alpha_2$.

On a alors

$$\begin{aligned}
\mathcal{L}(\mathsf{Product}_{\wedge}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) \\
\mathcal{L}(\mathsf{Product}_{\vee}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2) \\
\mathcal{L}(\mathsf{Product}_{\rightarrow}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathcal{L}(\mathcal{A}_1)^C \cup \mathcal{L}(\mathcal{A}_2) \\
\mathcal{L}(\mathsf{Product}_{\leftrightarrow}(\mathcal{A}_1, \mathcal{A}_2)) &= (\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)) \cup (\mathcal{L}(\mathcal{A}_1)^C \cap \mathcal{L}(\mathcal{A}_2))^C \\
\mathcal{L}(\mathsf{Product}_{\oplus}(\mathcal{A}_1, \mathcal{A}_2)) &= (\mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}_2)) \cup (\mathcal{L}(\mathcal{A}_2) \setminus \mathcal{L}(\mathcal{A}_1))
\end{aligned}$$

Enfin, introduisons la définition d'automate suspendable.

Définition 55 (Automate suspendable, [4, lemme 1])**.** *Un automate $\mathcal{A}$ sur un alphabet Σ est qualifié de* suspendable *si $\mathcal{L}(\mathcal{A}) = \Sigma^* \cdot \mathcal{L}(\mathcal{A})$, c'est-à-dire que l'ajout d'un préfixe fini à un mot ne change pas son acceptation.*

Pour obtenir un automate à partir d'une formule LTL, diverses méthodes existent et sont implémentées dans plusieurs outils. On peut par exemple citer `delag` [57] [2] ou `ltl3tela` [51].

Pour le premier, un découpage de la formule LTL est fait de manière à obtenir des sûretés syntaxiques, des garanties syntaxiques, des formules d'équité, et des formules n'entrant dans aucune de ces catégories.

Pour chaque classe une construction spécifique est utilisée et les automates associés à chaque sous-formule sont ensuite assemblés à l'aide d'un produit. Même si nous n'allons pas entrer davantage dans les détails, la structure des sous-formules est également utilisée pour optimiser ce produit.

Pour `ltl3tela` [51], l'idée de `delag` de découper en sous-formules puis de faire le produit des TELA associés à ces sous-formules est conservée mais l'approche diffère au niveau de la traduction des formules LTL. En effet, plusieurs traductions ont lieu et le plus petit des automates obtenus est conservé. De manière globale ces traductions sont faites sur la formule d'entrée avec une procédure utilisant des automates dits "alternants" et avec `ltl2tgba` (Spot) mais aussi sur la négation de la formule. De plus une déterminisation est aussi effectuée si besoin.

Même si Major et al. [51] ont montré que `ltl3tela` permet effectivement l'obtention de TELA plus petits, le nombre important de traductions engendre une durée de traitement supérieure aux autres outils.

Une dernière construction à mettre en avant est celle qui était utilisée jusqu'en 2020 par `Strix` [50] pour construire directement un automate de parité à partir d'une formule

2. `delag` désigne Deterministic Emerson-Lei Automata Generator

LTL. L'idée est de décomposer la formule de départ en un ensemble de sous-formules et d'associer à chacune d'entre elles des propriétés que porteront les automates qui leur seront associés (telle que la condition d'acceptation). L'étude de ces propriétés donne alors une méthode pour combiner les sous-automates pour obtenir un automate de parité.

Terminons cette liste non exhaustive par `ltl2tgba` qui est l'outil de traduction associé à `ltlsynt` pour la traduction de formule LTL en TELA. Même s'il s'agit d'une description plus poussée que celle des autres outils, cela restera cependant une simplification du processus de traduction.

Le processus utilisé par `ltl2tgba` s'appuie sur la transformation effectuée par `delag`. Il s'agit en effet de réécrire la formule sous forme normale négative avant de découper la formule et de recombiner les sous-automates issus de la traduction de chacune de ces sous-formules.

Cette construction s'appuie sur plusieurs procédures :

Product$_\odot$ (voir le théorème 3) construit le produit de deux automates déterministes à l'aide d'un produit synchronisé standard et combine leur condition d'acceptation en utilisant $\odot \in \{\wedge, \vee, \leftrightarrow, \oplus\}$;

ProductSusp$_\odot$ construit également un produit mais suppose que le second automate est associé à une propriété suspendable, ce qui implique que le produit n'a à être fait qu'avec les composantes fortement connexes acceptantes du premier automate. Il s'agit d'une méthode similaire à celles décrites par Müller et Sickert [57] et par Babiak et al. [4] ;

BuildMinWDBA s'appuie sur la capacité de Spot à produire un automate de Büchi déterministe faible minimal pour toute formule d'obligation [18].

GFGuaranteeToDBA est un algorithme inspiré de celui décrit par Esparza et al. [24] et permettant d'extraire un automate de Büchi déterministe d'une formule de la forme $\mathsf{G}(\bigwedge_i \mathsf{F}(\alpha_i))$ où les α_i sont des garanties syntaxiques ;

Complement (voir la définition 41) dualise la condition d'acceptation d'un TELA déterministe pour le complémenter ;

ToNBA transforme une formule LTL en un automate de Büchi non-déterministe [20] ;

Determinize déterminise un automate non-déterministe en automate de parité à l'aide de l'algorithme de Redziejowski [64].

4.3. Autres outils existants

Décrivons maintenant un ensemble d'outils permettant de résoudre le problème de la synthèse LTL. Le but de cette section n'est pas de décrire en détail leur principe mais de donner une idée de différentes méthodes utilisées.

Algorithme 1 : Traduction de formule LTL en automate d'Emerson-Lei déterministe

Entrée : Une formule LTL ϕ sous forme normale négative et un opérateur binaire optionnel *op*.

Sortie : Un automate d'Emerson-Lei déterministe.

Function ToDELA(ϕ, $op = \bot$)
  si ϕ *est de la forme* $\underbrace{\mathsf{X}\mathsf{X}\ldots\mathsf{X}}_{i \text{ copies de } \mathsf{X}}\ \alpha$ **alors**
    $\mathcal{A}_\alpha \leftarrow$ ToDELA(α)
    créer $\mathcal{A}$ en ajoutant une chaîne de i états avant l'état initial de $\mathcal{A}_\alpha$
    retourner $\mathcal{A}$
  si ϕ *est de la forme* $f_1 \odot \cdots \odot f_n$ *pour* $\odot \in \{\wedge, \vee, \leftrightarrow, \oplus\} \setminus \{op\}$ *et* $n \geq 2$ **alors**
    Partitionner $\{f_1, \ldots, f_n\}$ comme $S \uplus O \uplus R$ où
    $\begin{cases} S : \textit{formules suspendables} \\ O : \textit{obligation formulas} \\ R : \text{autres formules} \end{cases}$
    $\mathcal{A} \leftarrow \mathsf{Product}_\odot(\mathsf{ToDELA}(\bigodot_{f \in R} f, \odot), \mathsf{BuildMinWDBA}(\bigodot_{f \in O} f))$
    pour $s \in S$ **faire**
      $\mathcal{A} \leftarrow \mathsf{ProductSusp}_\odot(\mathcal{A}, \mathsf{ToDELA}(s, \odot))$
    retourner $\mathcal{A}$
  si ϕ *est une obligation syntaxique* **alors**
    retourner BuildMinWDBA(ϕ)
  si ϕ *est de la forme* $\mathsf{G}(\bigwedge_i \mathsf{F}\alpha_i)$ *où les* α_i *sont des garanties syntaxiques* **alors**
    retourner GFGuaranteeToDBA(ϕ)
  si ϕ *est de la forme* $\mathsf{F}(\bigvee_i \mathsf{G}\alpha_i)$ *où les* α_i *sont des sûreté syntaxique* **alors**
    retourner Complement(GFGuaranteeToDBA($\neg\phi$))
  retourner Determinize(ToNBA(ϕ))
retourner Determinize(ToNBA(ϕ))

4.3.1. BoSy

BoSy [27] [3] est un outil développé par Faymonville et al. s'appuyant sur la *bounded synthesis* garantissant la minimalité du résultat.

L'idée est de produire à partir d'un automate associé à la spécification des contraintes logiques donnant une borne sur la taille de la solution et de confier à un solveur de contraintes la résolution de ce problème. Si ces contraintes peuvent être satisfaites, alors une solution est construite, sinon la limite de taille est augmentée.

Une des particularités de BoSy est qu'il permet de faire plusieurs tâches en parallèle. Par exemple il est possible de chercher à la fois une stratégie pour le contrôleur (joueur 1) sur la spécification et une stratégie pour l'environnement (joueur 0) sur la négation de celle-ci.

L'outil s'appuie sur `ltl3ba` ou `Spot` pour obtenir un automate de co-Büchi universel qui sert ensuite à encoder le problème sous formes de contraintes (SAT, SMT, QBF, DQBF/EPR).

4.3.2. SPORE

À la différence de `ltlsynt`, le but de `SPORE` est de produire un automate de parité généralisée. C'est-à-dire une conjonction ou une disjonction de conditions de parité.

Pour cela la condition est découpée en un ensemble de sous-formules qui sont séparément traduites en automates de parité. Le produit de ces automates est un automate de parité généralisée.

Un tel jeu peut être résolu par une adaptation de l'algorithme de Zielonka [17].

4.3.3. Strix

Strix [50] est un outil visant à résoudre deux problèmes rencontrés par des outils tels que `ltlsynt`.

Le premier est l'obligation de construire un automate de parité avant de chercher une stratégie gagnante. Strix combine ces deux étapes, ce qui permet d'arrêter la construction de l'automate si un des deux joueurs possède une stratégie depuis l'état initial.

Le deuxième problème concerne la complexité des formules qui doivent être traitées.

Jusqu'en 2020 l'outil s'appuyait sur l'idée qu'il est possible de découper une formule en un ensemble de sous-formules simples à traduire. Un automate de parité est ensuite obtenu en combinant les automates issus de la traduction de ces sous-formules.

Depuis 2021 une étape de réécriture a été ajoutée pour mettre la spécification sous forme normale Δ_2. Cela permet de découper la formule en sous-formules de la forme Π_2 et Σ_2 qui peuvent être facilement transformées en automates de Büchi et co-Büchi. La combinaison de ces automates permet d'obtenir un TELA qui est ensuite paritisé à l'aide d'une procédure combinant ACD et les arbres de Zielonka.

3. Le dépôt associé à BoSy est `https://github.com/reactive-systems/bosy`

5. Combinaison de procédures de paritisation

Dans ce chapitre, notre but est de produire un automate de parité (PA dans la figure 5.1) à partir d'un automate à condition quelconque.

La figure 5.1 présente les différents moyens de réaliser cette transformation qui seront décrits dans ce chapitre.

Ces transformations ont été partiellement décrites dans « Practical "Paritizing" of Emerson-Lei Automata » [65]. Cependant depuis cette publication d'autres transformations ont été introduites et l'algorithme général a été réimplémenté de manière à diminuer le temps de traitement.

Nous commencerons par présenter les algorithmes CAR, TAR et SAR permettant de traduire tout type d'automate en automate de parité.

Un algorithme nommé IAR qui en est proche se limite uniquement au traitement des automates de Rabin ou Streett et sera également présenté. Il vise à s'appuyer sur la structure particulière de ces conditions pour produire un automate de parité plus petit qu'avec les algorithmes généraux.

Un autre algorithme permettant l'obtention d'un automate de parité (Büchi précisément) est la dégénéralisation. Même si elle ne sera pas évaluée, cette méthode sera présentée et servira à introduire la dégénéralisation partielle.

L'idée qu'il peut être possible de recolorer un automate de manière à obtenir un automate de Büchi (resp. parité) nous conduira à la présentation d'une méthode permettant de détecter ces automates dits Büchi-type (resp. parity-type).

Une procédure de paritisation que nous désignerons par `to_parity` (son nom dans Spot) regroupera ces différents algorithmes tout en y apportant diverses heuristiques.

Il sera ensuite présenté une transformation s'appuyant sur les arbres de Zielonka donnant une certaine notion d'optimalité par rapport à CAR, TAR, IAR ou encore SAR.

Une étude de l'impact des différentes optimisations de `to_parity` ainsi qu'une comparaison avec l'utilisation des arbres de Zielonka sera effectuée.

5.1. Méthode générale de paritisation

Avant de lister différents algorithmes de paritisation, mettons en avant que pour une grande partie d'entre eux (CAR, IAR, TAR, SAR, dégénéralisation, dégénéralisation partielle, ACD et arbres de Zielonka) il s'agit d'une construction associant un état q de l'automate de départ à une mémoire m, formant des états (q, m). Cette méthode commune correspond à l'algorithme 2.

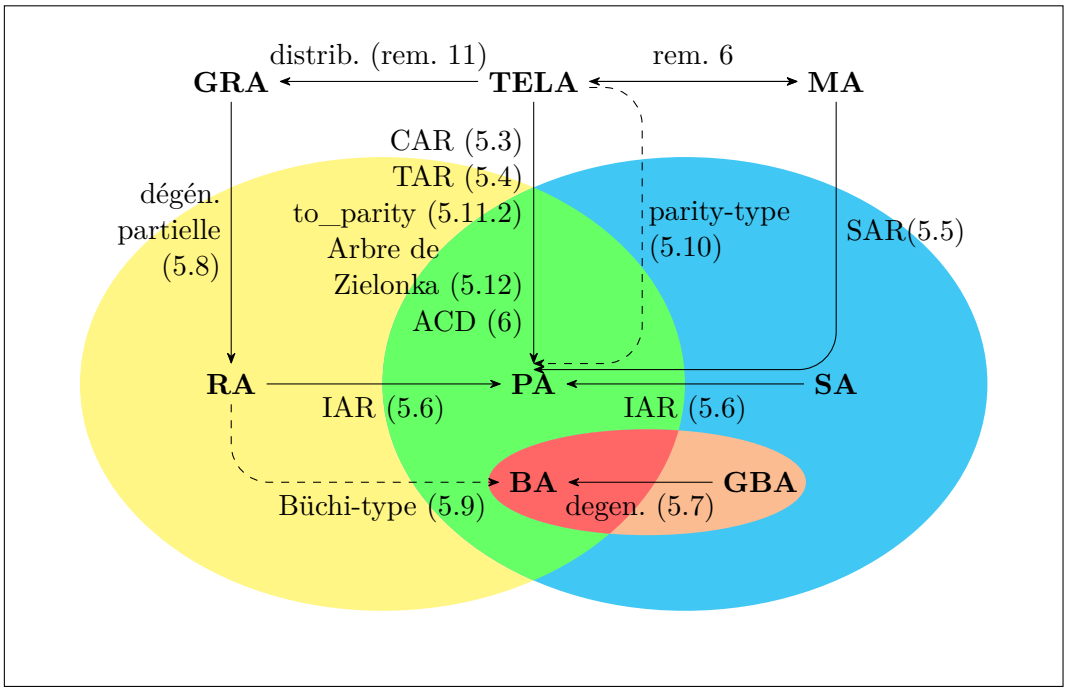


FIGURE 5.1. – Différentes classes de transformation d'automate en automate de parité. Les ensembles portent des classes de condition d'acceptation. Deux types d'automates sont dans la même classe s'il est possible de passer d'une condition à une autre sans changer la structure des transitions. Une flèche entre deux classes porte des algorithmes permettant de créer un automate de la classe de destination à partir d'un automate de la classe de départ. Les flèches en pointillées indiquent qu'il s'agit de transformations préservant la structure de l'automate (recoloration) ne pouvant être appliquées que sur certains automates de la classe. GRA correspond aux automates de Rabin généralisés, TELA aux automates d'Emerson-Lei, RA aux automates de Rabin, PA aux automates de parité, SA aux automates de Streett, BA aux automates de Büchi, GBA aux automates de Büchi généralisés et MA aux automates de Muller.

Algorithme 2 : Structure commune aux algorithmes LAR, de dégénéralisation, ACD et arbres de Zielonka

Entrée : Un automate $\mathcal{A}$ avec une condition α sur les états de $\mathcal{A}$
Sortie : Un automate de parité $\mathcal{P}$ équivalent à $\mathcal{A}$

1 Initialisation
`/* Création de l'état initial du résultat */`
2 $(q_0, m_0) \leftarrow$ Crée l'état initial
3 $Q' \leftarrow \{(q_0, m_0)\}$
`/* L'automate n'a pas de transition */`
4 $\delta' \leftarrow \{\}$
5 `todo` $\leftarrow \{(s, m)\}$
6 **tant que** *`todo` n'est pas vide* **faire**
7 $(s, m) \leftarrow$ `todo.top()`
8 `todo.pop()`
9 **pour chaque** $e \in$ *`edges(s)`* **faire**
10 Crée un nouvel état(q', m') si besoin
11 ajouter (q', m') à `todo` s'il n'existait pas déjà
12 Crée une nouvelle arête
13 Assigne la condition au résultat
14 **retourner** $\mathcal{P}$

L'idée est d'initialiser la construction avec une paire associant l'état initial de l'automate de départ à une mémoire pouvant dépendre de cet état. Il s'agit ensuite d'effectuer un parcours en profondeur, donc d'avoir une pile `todo` contenant les états du résultat qui doivent être traités. Pour chaque tel état (q, m) on regarde les arêtes quittant q dans l'automate de départ et on les traite pour créer de nouveaux états et une nouvelle arête. Enfin en fonction de l'algorithme, une condition de parité est assignée.

5.2. Principe des algorithmes de type LAR

Les algorithmes de type *LAR* (Later Appearance Record) ont pour but de construire un automate de parité à partir d'un ω-automate. Ce dernier peut être d'Emerson-Lei, de Muller, ou alors restreint aux conditions de Rabin par exemple. Ils s'appuient tous sur l'utilisation d'une mémoire portant certaines parties d'un automate (états, couleurs, arêtes...). L'idée est que durant une exécution, mémoriser si ces éléments sont vus infiniment souvent permet de savoir si cette exécution est acceptante ou non. LAR s'appuie sur les modifications de cette mémoire durant l'exécution pour attribuer une condition de parité.

Des algorithmes tels que LAR ou l'utilisation des arbres de Zielonka peuvent être vus comme la composition de deux transducteurs. D'un côté une machine produisant les couleurs sur l'automate de départ et de l'autre une machine prenant ces couleurs et

produisant la couleur que devra porter le résultat.

5.3. Color Appearance Record (Contribution)

Le premier algorithme de la famille LAR que nous allons étudier est *Color Appearance Record* (CAR). Il s'agit d'une généralisation de SAR que nous montrerons par la suite.

Décrivons de manière formelle cet algorithme. Posons-nous le cadre de la transformation d'un automate à condition quelconque $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ en un automate de parité $\mathcal{A}^{\mathrm{CAR}} = (Q^{\mathrm{CAR}}, M^{\mathrm{CAR}}, \delta^{\mathrm{CAR}}, q_0^{\mathrm{CAR}}, \alpha^{\mathrm{CAR}})$.

On désignera par *historique* un ordre total sur les éléments de M. L'ensemble de ces ordres totaux est noté $\Pi(M)$.

L'ensemble $\mathsf{Seq}(M)$ désignera l'ensemble des suites ordonnées des éléments de tout sous-ensemble de M.

Nous utiliserons une fonction $\mathcal{U} : \Pi(M) \times \mathsf{Seq}(M) \to \Pi(M) \times \mathbb{N}$ déplaçant la séquence d'éléments de M à gauche de l'historique donné et indiquant la plus grande position à laquelle se trouvait une telle couleur. Elle est définie de manière récursive par :

$$\mathcal{U}(\sigma, ()) = (\sigma, 0)$$
$$\mathcal{U}(\sigma, (c_1, \ldots, c_k)) = (\sigma', i)$$

où $\sigma' = \langle c_1, \ldots, c_k \rangle \cdot \pi$ et π correspond à σ privé de $\{c_1, \ldots, c_k\}$ alors que $i = 1 + \max(\sigma^{-1}(c_1), \ldots, \sigma^{-1}(c_k))$.

De plus nous utiliserons une fonction de coloration d'arête $\kappa : \Pi(M) \times \mathbb{N} \to M'$ définie par

$$\kappa(\sigma, i) = \begin{cases} 2i & \text{si } \{\sigma(0), \ldots, \sigma(i-1)\} \vDash \alpha \\ 2i+1 & \text{sinon.} \end{cases}$$

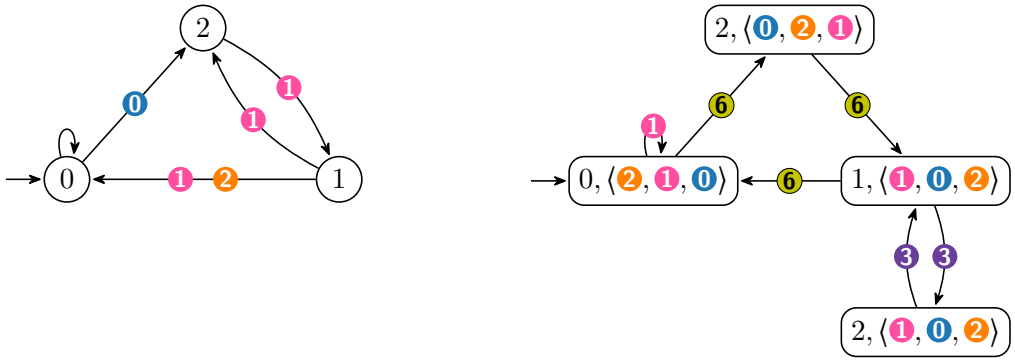
On définit l'automate CAR de $\mathcal{A}$ de la manière suivante :

Définition 56 (Automate CAR)**.** *Pour un automate $\mathcal{A}$ tel que décrit précédemment, on définit l'automate CAR $\mathcal{A}^{CAR} = (Q^{CAR}, M^{CAR}, \Sigma, \delta^{CAR}, q_0^{CAR}, \alpha^{CAR})$ comme :*

- $Q^{CAR} \subseteq Q \times \Pi(M)$;
- $M^{CAR} = \{0, \ldots, 2|M| + 1\}$;
- α^{CAR} *est une condition de parité maximale paire ;*
- $q_0^{CAR} = (q_0, \pi)$ *pour un élément π quelconque de $\Pi(M)$;*

et $\delta^{CAR} \subseteq Q^{CAR} \times \Sigma \times 2^{M^{CAR}} \times Q^{CAR}$ est telle que

$$\delta^{CAR} = \left\{ (q, \pi) \xrightarrow{x, \{c\}} (q', \sigma') \;\middle|\; q \xrightarrow{x, C} q' \in \delta,\ (\sigma', i) = \mathcal{U}(\sigma, C),\ c = \kappa(\sigma', i) \right\}$$



(a) Automate à paritiser

(b) Automate de parité associé à l'automate de la figure 5.2a

FIGURE 5.2. – Exemple d'application de CAR

Exemple 10. *Montrons comment est construit l'automate de parité de la figure 5.2b à partir de celui de la figure 5.2a en utilisant CAR.*

La première étape est la création d'un état $(0, \langle 2, 1, 0 \rangle)$ *associant l'état initial de l'automate de départ et une mémoire quelconque.*

Traitons alors la boucle de l'état 0. Celle-ci ne porte aucune couleur. Ainsi $\mathcal{U}(\langle 2, 1, 0 \rangle, ()) = (\langle 2, 1, 0 \rangle, 0)$*. Puisque l'ensemble de couleurs vide n'est pas acceptant pour* $(\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge \mathsf{Inf}(2)$*, alors le résultat de la fonction* κ *sera* $2 \times 0 + 1 = 1$*.*

Considérons maintenant le passage de l'état $(1, \langle 1, 0, 2 \rangle)$ *à* $(0, \langle 2, 1, 0 \rangle)$*.*

Puisque les couleurs 1 *et* 2 *sont présentes, on décide de déplacer* 1 *puis* 2*. La plus grande position associée à une de ces couleurs dans* $\langle 1, 0, 2 \rangle$ *est* 3 *et puisque l'ensemble* $\{0, 1, 2\}$ *des couleurs dont la position change dans la mémoire est acceptant, alors la couleur produite est* $2 \times 3 + 0 = 6$*.*

Correction de CAR

Montrons maintenant que l'automate produit avec CAR reconnaît le même langage que l'automate que cette procédure prend en entrée. Notre preuve s'appuie sur celle faite par LÖDING [47] pour montrer la correction de SAR. Ce sera également l'occasion d'introduire les définitions de séparation et visualisation qui nous seront utiles pour décrire SAR.

Pour cela nous nous appuierons sur la définition d'exécution CAR associée à un automate.

Définition 57 (Exécution CAR)**.** *Soit* $r = (s_i \xrightarrow{\ell_i, A_i} s_{i+1})_{i \in \mathbb{N}}$ *une exécution de* $\mathcal{A}$ *et* $\sigma \in \Pi(M)$*. On définit de la manière suivante l'ensemble* $\rho_{CAR}(r, \sigma) \subseteq (Q^{CAR} \times \Sigma \times$

Algorithme 3 : Algorithme de construction CAR

Entrée : Un automate $\mathcal{A}$ avec une condition α sur les états de $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$, un historique initial m_0
Sortie : Un automate de parité $\mathcal{P}^{\text{CAR}}$ équivalent à $\mathcal{A}$

```
   /* Création de l'état initial du résultat                          */
 1 Q' ← {(q_0, m_0)}
 2 δ' ← {}
 3 todo ← {(q_0, m_0)}
 4 tant que todo n'est pas vide faire
 5 |  (q, m) ← todo.top()
 6 |  todo.pop()
 7 |  pour chaque arête e = (q, ℓ, c, q') partant de q faire
   |  |  /* Plus grande position d'une couleur déplacée              */
 8 |  |  max_pos ← 0
 9 |  |  pour col : c faire
   |  |  |  /* max_pos récupère la plus grande position où se
   |  |  |     trouvait une des couleurs dans c                      */
10 |  |  |  max_pos ← max(max_pos, 1 + m^{-1}(col))
   |  |  |  /* Déplacement de la couleur à gauche de l'historique    */
11 |  |  |  m' ← col · (m ∖ col)
   |  |  /* Calcul de la couleur. ⊭ α vaut 1 si l'ensemble des
   |  |     couleurs de associé vérifie α                             */
12 |  |  c' ← 2 × |cols| + (m(0), ..., m(max_pos − 1)) ⊭ α
13 |  |  si (q', m') ∉ Q' alors
14 |  |  |  todo.append((q', m'))
15 |  |  |  Q' ← Q' ∪ {(q', m')}
16 |  |  δ' ← δ' ∪ {((q, m), ℓ, c', (q', m'))}
17 retourner (Q', {0, ..., 2 × |M| + 1}, Σ, δ', (q_0, m_0), parité maximale paire})
```

$2^{M'} \times Q^{CAR})^\omega$ *des exécutions CAR associés à l'exécution* r *et l'historique initial* σ*. Une exécution* $r' = (s'_i \xrightarrow{\ell'_i, B_i} s'_{i+1})_{i \in \mathbb{N}}$ *appartient à* $\rho_{CAR}(r, \sigma)$ *si et seulement si* $\forall i \in \mathbb{N}$ *:*

- $\ell_i = \ell'_i$ *(les deux exécutions ont les mêmes étiquettes) ;*
- $s'_i = (s_i, \sigma_i)$ *(*r' *associe à un état de* r *un historique) ;*
- $\sigma_0 = \sigma$ *(l'historique initial est la permutation* σ*) ;*
- *il existe un ordre* $(a_1, \ldots, a_k) \in \mathsf{Seq}(M)$ *de* A_i *et un indice* $\iota_{i+1} \in M$ *tel que* $\mathcal{U}(\sigma_i, (a_1, \ldots, a_k)) = (\sigma_{i+1}, \iota_{i+1})$ *(l'historique est mis à jour en accord avec les couleurs vues par* r*) ;*
- $B_i = \{\kappa(\sigma_{i+1}, \iota_{i+1})\}$ *(toutes les arêtes sont colorées en accord avec la condition d'acceptation et la taille de l'ensemble des couleurs déplacées).*

Remarque 25. *L'existence de plusieurs éléments dans* ρ_{CAR} *est liée à la possibilité de*

trouver plusieurs couleurs sur une arête de $\mathcal{A}$. Dans le cas où plusieurs couleurs peuvent se trouver sur une arête, de multiples exécutions CAR existent et diffèrent sur l'ordre des couleurs déplacées.

Définition 58 (Historique visualisant un ensemble)**.** *On dit qu'un historique σ visualise un ensemble $F \subseteq M$ s'il existe un indice $i \in \mathbb{N}$ tel que $F = \{\sigma(0), \dots, \sigma(i-1)\}$, c'est-à-dire si F contient exactement les i premiers éléments de σ. On peut également dire σ sépare F à l'indice i.*

La définition de la fonction de mise à jour $\mathcal{U}$ implique le lemme suivant. De manière intuitive il indique dans une première partie que la dernière couleur insérée se situe à gauche de l'historique après la mise à jour alors que la seconde partie indique que lorsqu'une couleur c est déplacée à gauche de l'historique, l'ensemble des couleurs qui sont déplacées correspond exactement à l'ensemble des couleurs vues depuis la précédente occurrence de c.

Lemme 1. *Soit $\sigma \in \Pi(M)$ un historique, $C \subseteq M$ un ensemble de couleurs, $k \in \mathbb{N}^*$ et $(C_0, \dots, C_k) \in \mathsf{Seq}(M)^k$ tel que $C_0 \cup \dots \cup C_k = C$. Posons $\sigma_0 = \sigma$ et $\mathcal{U}(\sigma_i, C_i) = (\sigma_{i+1}, \iota_{i+1})$ pour $i \in \{0, \dots, k\}$. Alors :*

1. *σ_{k+1} visualise C ;*
2. *Si σ visualise C, alors il existe $\ell \in \{0, \dots, k\}$ tel que $\sigma_{\ell+1}$ sépare C à l'indice $\iota_{\ell+1}$.*

Démonstration. Le premier point du lemme est évident par définition de $\mathcal{U}$: les dernières couleurs vues (C) sont insérées à gauche de l'historique.

Prouvons alors le second point du lemme. Pour tout $c \in C$, définissons $\mu(c) = \min\{m \mid c \in C_m\}$ la première occurrence de la couleur c dans les ensembles de couleurs. Soient $\ell = \max\{\mu(c) \mid c \in C\}$ et $L = \{c \in C \mid \mu(c) = \ell\}$ étant respectivement l'indice et l'ensemble des couleurs qui ont été les derniers à être vus.

Notons que si nous mettons à jour un historique σ visualisant C en insérant des couleurs appartenant à C, alors l'historique résultant continuera de visualiser C. Pour $i \in \{0, \dots, k+1\}$, σ_i visualise donc C.

Ainsi, σ_ℓ doit être de la forme $\pi \cdot \pi_L \cdot \pi'$ où π, π_L et π' sont respectivement des permutations de $C \setminus L$, L et $M \setminus C$. Intuitivement, les couleurs de $C \setminus L$ ont été déplacées à gauche de σ_ℓ qui continue de visualiser C. Soit $j = |C| - 1$ l'indice de l'élément de L le plus à droite dans σ_ℓ. De manière évidente, $L \subseteq C_\ell$. Alors par définition de $\mathcal{U}$, $\iota_{\ell+1} = j + 1$. Puisque $\sigma_{\ell+1}$ visualise C, alors il le sépare également à l'indice $\iota_{\ell+1}$. □

Définition 59 (Exécution séparant un ensemble)**.** *On dit qu'une exécution r sépare un ensemble F s'il existe un indice $i \in \mathbb{N}$ tel que σ_i sépare F à l'indice ι_i.*

F est séparé infiniment souvent s'il existe un nombre infini de tels indices i.

De manière intuitive, un ensemble est séparé par une exécution r s'il est exactement l'ensemble des éléments déplacés après avoir inséré quelques couleurs dans un des historiques de l'exécution CAR. Le lemme suivant signifie que les seuls ensembles séparés infiniment souvent par une exécution CAR sont soit Rep soit ses sous-ensembles.

Lemme 2. *Soit r une exécution d'un TELA, σ un historique et $r' \in \rho_{CAR}(r, \sigma)$.*

1. $\mathsf{Rep}(r)$ *est séparé infiniment souvent par* r' ;
2. *Si* r' *sépare* F *infiniment souvent, alors* $F \subseteq \mathsf{Rep}(r)$.

Démonstration. On sait qu'il existe un rang i_0 à partir duquel les seules couleurs associées aux arêtes de r correspondent à $\mathsf{Rep}(r)$. De plus par définition de Rep, il existe un rang $i_1 \geq i_0$ tel que $A_{i_0} \cup \ldots \cup A_{i_1} = \mathsf{Rep}(r)$. Alors d'après le lemme 1.1 et par définition de ρ_{CAR}, σ_{i_1} visualise $\mathsf{Rep}(r)$ peu importe l'ordre d'insertion des couleurs $A_{i_0}, \ldots, A_{i_1}$ associées aux arêtes.

Par définition de Rep, il existe également un rang $i_2 \geq i_1$ tel que $A_{i_1} \cup \ldots \cup A_{i_2} = \mathsf{Rep}(r)$. Alors par le lemme 1.2 et la définition de ρ_{CAR}, il existe un indice $i_1 \geq u_0 \geq i_2$ tel que σ_{u_0} sépare σ_{u_0} à l'indice ι_{u_0}. De manière plus générale, on peut calculer une suite $(u_i)_{i \in \mathbb{N}}$ telle que σ_{u_i} sépare $\mathsf{Rep}(r)$ à l'indice ι_i. Ainsi le lemme 2.1 est vérifié.

De plus, puisque σ_{u_0} sépare exactement $\mathsf{Rep}(r)$ à l'indice ι_{u_0} et les seules couleurs ajoutées après le rang u_0 correspondent à $\mathsf{Rep}(r)$, par définition de la $\mathcal{U}$, $\forall i \geq u_0$, σ_i peut seulement séparer un sous-ensemble de $\mathsf{Rep}(r)$ à l'indice ι_i. Ainsi le lemme 2.2 est vérifié. □

Le lemme suivant indique que $\mathsf{Rep}(r)$ est le exactement le plus grand ensemble séparé par une exécution CAR r'.

Lemme 3. *Soit r une exécution d'un TELA, $r' \in \rho_{CAR}(r, \sigma)$ et $F \subseteq M$. Alors $\mathsf{Rep}(r) = F$ si et seulement si :*

1. F *est séparé infiniment souvent par* r' ;
2. *si* F' *est séparé infiniment souvent par* r', *alors soit* $F' = F$ *soit* $|F| > |F'|$.

Démonstration. Supposons que $\mathsf{Rep}(r) = F$. Le premier point du lemme est vérifié par le lemme 2.1 alors que le second est vérifié par le lemme 2.2.

Soit $F \subseteq M$ vérifiant les deux points du lemme 3. D'après le lemme 2.2, $F \subseteq \mathsf{Rep}(r)$. Cependant, d'après le lemme 2.1, $\mathsf{Rep}(r)$ est séparé infiniment souvent par r'. Ainsi soit $\mathsf{Rep}(r) = F$ soit $|F| > |\mathsf{Rep}(r)|$. Puisque $F \subseteq \mathsf{Rep}(r)$, on a forcément $\mathsf{Rep}(r) = F$. □

Notre preuve de correction de la construction CAR s'appuie sur le théorème suivant :

Théorème 4. *Soient r une exécution d'un TELA $\mathcal{A}$ portant une condition α, $\sigma \in \Pi(M)$ et $r' \in \rho_{CAR}(r, \sigma)$. Alors $r \vDash \alpha$ si et seulement si $r' \vDash \alpha'$ où α' est une condition de parité maximale paire.*

Démonstration. D'après le lemme 3, $\mathsf{Rep}(r)$ est le plus grand ensemble séparé infiniment souvent par r'. Ainsi, si $\mathsf{Rep}(r) \vDash \alpha$, alors la plus grande couleur visitée infiniment souvent est $2 \cdot |\mathsf{Rep}(r)|$, qui est paire. De plus si $\mathsf{Rep}(r) \nvDash \alpha$, alors la plus grande couleur vue infiniment souvent par r' est $2 \cdot |\mathsf{Rep}(r)| + 1$ qui est impaire. Donc r est acceptant si et seulement si r' est associé à une condition de parité maximale paire. □

Il nous reste maintenant à relier ce théorème à l'automate construit.

Théorème 5 (Correction et conservation du déterminisme par la construction CAR)**.** *Soit $\mathcal{A}$ un TELA et $\mathcal{A}'$ l'automate de parité obtenu en appliquant CAR sur $\mathcal{A}$. Alors $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ et si $\mathcal{A}$ est déterministe, alors $\mathcal{A}'$ l'est également.*

Démonstration. Par construction, à chaque exécution r de $\mathcal{A}$, on peut lui faire correspondre une exécution r' de $\mathcal{A}'$ telle que $r' \in \rho_{\text{CAR}}(r, \sigma_0)$. D'après le théorème 4, si r est acceptant, il en est de même pour r'. Ainsi $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.

De plus à chaque transition de δ', on peut associer une transition dans δ. Ainsi à chaque exécution r' de $\mathcal{A}'$ on peut associer une exécution r de $\mathcal{A}$. D'après le théorème 4, si r est acceptant, alors il en est de même pour r'. Ainsi $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$.

Ainsi $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ et la définition de δ' préserve le déterminisme. □

Optimalité de LAR

Nous allons maintenant montrer l'optimalité des constructions de type LAR. Pour cela nous décrirons une famille d'automates de Streett tel que tout automate de Rabin équivalent porte un nombre d'état correspondant à la borne supérieure décrite. Puisqu'une condition de parité n'est qu'un cas particulier de condition de Rabin, cela nous donnera également le résultat pour la création d'automate de parité.

Par définition, l'automate produit par IAR a au plus $n \cdot k!$ états où n est le nombre d'états de l'automate de départ et k le nombre de paires de Streett. Nous allons montrer que cette borne supérieure est atteinte et est optimale. Cependant, mettons en avant qu'il s'agit ici de montrer, entre autres, que le résultat est minimal pour cet exemple mais que la construction n'est pas toujours minimale.

Notre preuve s'appuie de nouveau sur les travaux de Löding [47] mais est adaptée aux TELA.

Pour cette preuve, adaptons la définition de Rep pour un mot et un ensemble d'états.

Notation 7. *Soit σ une exécution associée à un mot $(w_i)_{i\in\mathbb{N}}$ sur un TELA dont l'alphabet est Σ.*

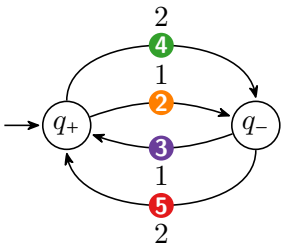
On note $\mathsf{Rep}_S(\sigma)$ l'ensemble des états visités infiniment souvent par σ.

On note $\mathsf{Rep}_\Sigma((w_i)_{i\in\mathbb{N}})$ l'ensemble des éléments de Σ vus infiniment souvent durant l'exécution σ.

Théorème 6. *Soit $n \in \mathbb{N}$, $n > 2$ et $\Sigma = \{1, \dots, n\}$. Considérons le langage $\mathcal{L}_n = \{(w_i)_{i\in\mathbb{N}} \in \Sigma^\omega \mid \mathsf{Rep}_\Sigma((w_{2i+1})_{i\in\mathbb{N}}) \subseteq \mathsf{Rep}_\Sigma((w_{2i})_{i\in\mathbb{N}})\}$ des mots infinis sur Σ tels que toute lettre apparaissant à des positions impaires infiniment souvent apparaît également infiniment souvent aux positions paires.*

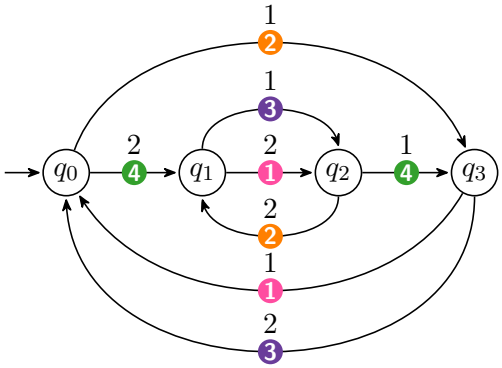
Il existe des automates de Streett déterministes ayant 2 états et n paires pour cette famille de langages mais tout automate de Rabin déterministe reconnaissant $\mathcal{L}_n$ a au moins $2 \times n!$ états.

Pour rappel, une condition de parité est une forme particulière de condition de Rabin, le théorème indique donc qu'il existe une famille d'automates pour lesquels IAR produit un automate minimal.



$\alpha = (\mathsf{Fin}(3) \vee \mathsf{Inf}(2)) \wedge (\mathsf{Fin}(5) \vee \mathsf{Inf}(4))$

(a) Automate $\mathcal{A}_2$



Parité maximale paire

(b) Automate $\mathcal{P}_2$

Un automate de Streett pour ce théorème est l'automate $\mathcal{A}_n = (Q, M_n, \Sigma, \delta_n, q_+, \alpha_n)$ où

— $Q = \{q_+, q_-\}$;

— $M_n = \{2, \dots, 2n+1\}$;

— $\delta_n = \{q_+ \xrightarrow{i,\{2i\}} q_- \mid i \in \Sigma\} \cup \{q_- \xrightarrow{i,\{2i+1\}} q_+ \mid i \in \Sigma\}$;

— $\alpha_n = \bigwedge_{i=1}^{i=n} \mathsf{Fin}(2i+1) \vee \mathsf{Inf}(2i)$;

et est représenté dans la figure 5.3a pour le cas $n = 2$.

Remarque 26. *Le langage $\mathcal{L}_n$ est stable par ajout ou suppression d'un préfixe de longueur paire.*

Démonstration. Prouvons par récurrence le théorème 6.

Si $n = 2$, nous considérons l'automate $\mathcal{A}_2$ de la figure 5.3a reconnaissant $\mathcal{L}_2$. Cet automate est équivalent à l'automate $\mathcal{P}_2$ de la figure 5.3b. Sa minimalité peut être vérifiée en utilisant l'algorithme de minimisation [3] implémenté dans Spot avec un SAT-solver. Puisque $\mathcal{P}_2$ possède 4 états, la propriété est vraie pour $n = 2$.

Supposons maintenant que le théorème est vrai au rang $n-1$ et prouvons qu'il est vrai au rang n. Soit $\mathcal{R} = (Q, M, \Sigma, \delta, q_0, \alpha)$ un automate de Rabin acceptant $\mathcal{L}_n$. Montrons qu'il existe n exécutions de $\mathcal{R}$ visitant au moins $2 \times (n-1)!$ états infiniment souvent sans qu'aucune de ces exécutions ne partage de tels états avec une autre.

Soit Q_{even} l'ensemble des états de Q accessibles depuis l'état initial q_0 en utilisant un nombre pair d'arêtes. Pour $q \in Q_{\text{even}}$ posons $\mathcal{R}_q = (Q, M, \Sigma, \delta, q, \alpha)$. La remarque 26 implique que $\mathcal{R}_q$ accepte $\mathcal{L}_n$. Notons que $\mathcal{L}_{n-1} = \mathcal{L}_n \cap (\Sigma \setminus \{n\})^\omega$. La restriction de $\mathcal{R}_q$ à l'alphabet $\Sigma \setminus \{n\}$ accepte donc $\mathcal{L}_{n-1}$. De la même manière, pour $i \in \{1, \dots, n\}$ la restriction $\mathcal{R}_q^i$ de $\mathcal{R}_q$ à l'alphabet $\Sigma_i = \Sigma \setminus \{i\}$ accepte un langage isomorphe à $\mathcal{L}_{n-1}$.

Par hypothèse de récurrence, R_q^i a au moins $2 \times (n-1)!$ états. Il a même une SCC avec au moins $2 \times (n-1)!$ états. En effet, considérons un état $q' \in Q_{\text{even}}$ qui appartient à une

des SCC terminales de $\mathcal{R}^i_q$. L'automate de Rabin $\mathcal{R}^i_{q'}$ accepte un langage isomorphe à $\mathcal{L}_{n-1}$ et continue de le faire si on supprime tous les états qui ne sont pas dans cette SCC terminale de $\mathcal{R}^i_q$. En appliquant l'hypothèse de récurrence sur cette restriction de $\mathcal{R}^i_{q'}$, on montre que cette SCC terminale de $\mathcal{R}^i_q$ a au moins $2 \times (n-1)!$ états. La fin de cette preuve s'appuie sur le lemme suivant :

Lemme 4. *Pour tout $i \in \{1, \dots, n\}$, il existe un mot infini $w^i \in {\Sigma_i}^\omega$ tel que :*

- *Il existe une exécution non-acceptante σ^i de $\mathcal{R}$ étiquetée par w^i ;*
- *Considérons l'ensemble $\mathsf{Rep}_S(\sigma^i)$ des états visités infiniment souvent par σ^i. Alors $|\mathsf{Rep}_S(\sigma^i)| \geq 2 \times (n-1)!$;*
- *Pour tout $j \in \Sigma_i$, $\mathsf{Rep}_S(\sigma^i) \cap \mathsf{Rep}_S(\sigma^j) = \varnothing$.*

Donc $\bigcup_{i=1}^{i=n} \mathsf{Rep}_S(\sigma^i) \subseteq Q$ et $\sum_{i=1}^{n} |\mathsf{Rep}_S(\sigma^i)| = n \times 2 \times (n-1)!$. On conclut alors que $|Q| \geq 2 \times n!$. □

Preuve du lemme 4. Soit $i \in \{1, \dots, n\}$ et q' un état de Q_{even} qui est dans une SCC terminale de $\mathcal{R}^i_{q_0}$. Soit $u \in \Sigma_i^*$ un mot permettant d'accéder à q' depuis q_0 dans $\mathcal{R}^i_{q_0}$. On a montré que $R^i_{q'}$ accepte $\mathcal{L}_{n-1}$. De plus il existe un mot $w \in \mathcal{L}_{n-1}$ tel que $\mathsf{Rep}_\Sigma((w_{2k+1})_{k\in\mathbb{N}}) = \mathsf{Rep}_\Sigma((w_{2k})_{k\in\mathbb{N}}) = \Sigma_i$. Il existe donc un mot $u' \in {\Sigma_i}^*$ tel que $u_0 = u \cdot u'$ est de longueur paire, contient toutes les lettres de Σ_i aux positions paires et aux positions impaires et visite au moins $2 \times (n-1)!$ états de $\mathcal{R}^i_{q_0}$. Soient $j \neq i$ et q_1 l'état atteint lors de la lecture de $u_0 ij$ dans $\mathcal{R}$. De manière similaire à u_0 il existe un mot $u_1 \in \Sigma_i^*$ tel que u_1 est de longueur paire, contient toute lettre de Σ_i aux positions paires et impaires, et visite au moins $2 \times (n-1)!$ états différents de $\mathcal{R}^i_{q_1}$. En répétant cette procédure on peut créer un mot infini $w^i = u_0 i j u_1 u j \dots$ tel que $\mathsf{Rep}_\Sigma((w^i_{2k+1})_{k\in\mathbb{N}}) = \Sigma$ et $\mathsf{Rep}_\Sigma((w^i_{2k})_{k\in\mathbb{N}}) = \Sigma_i$. Donc $w^i \notin \mathcal{L}_n$, $\mathcal{R}$ rejette w^i et le premier point du lemme est donc vérifié. De plus, soit σ^i l'exécution de $\mathcal{R}$ correspondant à w^i. Il existe un rang ℓ tel que σ^i ne visite que les états de $\mathsf{Rep}_S(\sigma^i)$ après ce rang. Cependant par construction le mot u_ℓ doit visiter au moins $2 \times (n-1)!$ états différents. Donc $\mathsf{Rep}_S(\sigma^i)$ contient au moins $2 \times (n-1)!$ états. Le deuxième point du lemme est alors vérifié.

Supposons maintenant qu'il existe $j \in \Sigma_i$ tel que $\mathsf{Rep}_S(\sigma^i) \cap \mathsf{Rep}_S(\sigma^j) \neq \varnothing$. Soit q un état de cette intersection. L'exécution infinie σ^i de $\mathcal{R}$ visite q infiniment souvent et on peut donc la découper en une suite infinie $(\sigma^i_k)_{k\in\mathbb{N}}$ d'exécutions finies de $\mathcal{R}$ telle que σ^i_0 amène à q depuis q_0 et pour tout $k \geq 1$, l'exécution non-triviale $(\sigma^i_k)_{k\in\mathbb{N}}$ permettant d'aller de q à q et de visiter toutes les couleurs de $\mathsf{Rep}(\sigma^i)$. On peut découper de manière similaire σ^j en une suite infinie de sous-suites $(\sigma^j_k)_{k\in\mathbb{N}}$ permettant d'aller de q à q en visitant toutes les couleurs de $\mathsf{Rep}(\sigma^j)$. Considérons alors l'exécution $\sigma = \sigma^i_0 \sigma^j_1 \sigma^i_1 \sigma^j_2 \sigma^i_2 \dots$ de $\mathcal{R}$. Cette exécution doit être rejetante car $\mathsf{Rep}(\sigma) = \mathsf{Rep}(\sigma^i) \cup \mathsf{Rep}(\sigma^j)$ et l'union de deux cycles rejetants d'un automate de Rabin est rejetant (voir page 28). Soit $w = (a_k)_{k\in\mathbb{N}}$ l'étiquette de σ. Par construction de σ on a $\mathsf{Rep}_S((a_{2k+1})_{k\in\mathbb{N}}) = \Sigma$ puisqu'il est soit égal à $\Sigma \cup \Sigma$, $\Sigma_i \cup \Sigma$, $\Sigma \cup \Sigma_j$ ou à $\Sigma_i \cup \Sigma_j$ en fonction de la préservation par σ de la parité des positions des lettres apparaissant infiniment souvent dans w^i et w^j. De manière similaire, $\mathsf{Rep}_S((w_{2k})_{k\in\mathbb{N}}) = \Sigma$. Donc $w \in \mathcal{L}_n$. Ce n'est pas possible car σ n'est pas une exécution

acceptante de $\mathcal{R}$. Ceci implique que $\mathsf{Rep}_S(\sigma^i) \cap \mathsf{Rep}_S(\sigma^j) = \varnothing$ et le troisième point du lemme est alors vérifié. □

5.4. Transition Appearance Record (Contribution)

Transition Appearance Record (TAR) est un variant de LAR utile dans le cas où il y a plus de couleurs ou de paires de Rabin/Streett que d'arêtes.

L'idée est qu'il est possible de décrire l'ensemble des couleurs vues pendant une exécution à partir de l'ensemble des transitions qui ont été vues. Ainsi l'*historique* peut ici contenir ces transitions et une transition vue sera déplacée au début de l'historique. En pratique nous mémoriserons plutôt les arêtes qui ont été vues pour limiter la taille de l'historique.

Nous allons ici travailler sur un TELA $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ associé à un graphe $G_{\mathcal{A}} = (V, E)$.

De manière plus formelle introduisons la fonction $\mathcal{U} : \Pi(E) \times E \to \Pi(E) \times \mathbb{N}$ déplaçant une arête à gauche d'un historique et indiquant l'indice où se trouvait cette arête. Elle est définie par

$$\mathcal{U}(\sigma, e) = (\sigma', i)$$

où $\sigma' = \langle e \rangle \cdot \pi$, π correspond à σ privé de e et $i = \sigma^{-1}(e)$.

Introduisons également la fonction de coloration d'une arête $\kappa : \Pi(E) \times \mathbb{N} \to M'$ définie par

$$\kappa(\sigma, i) = \begin{cases} 2i & \text{si } \Gamma(e) \vDash \alpha \\ 2i+1 & \text{sinon} \end{cases}$$

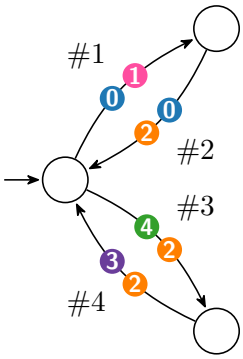
L'automate construit par la procédure TAR est alors défini comme

Définition 60 (Automate TAR)**.** *Pour un automate $\mathcal{A}$ tel que décrit précédemment, on définit l'automate TAR $\mathcal{A}^{TAR} = (Q^{TAR}, M', \Sigma, \delta, q_0', \alpha')$ comme :*

- $Q^{TAR} \subseteq Q \times \Pi(E)$;
- $M' = \{0, \dots, 2|E| + 1\}$;
- $q_0' \in \{q_0\} \times \Pi(E)$;
- α' *est une condition de parité maximale paire ;*

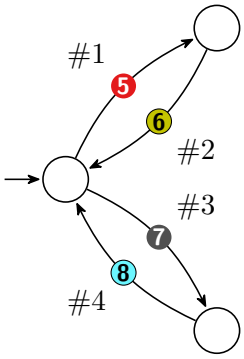
et δ' est définie de la manière suivante : Soient $e = (q, a, c, q') \in \delta$ et $(q, \sigma), (q', \sigma') \in Q^{TAR}$. alors $((q, \sigma), a, c', (q', \sigma')) \in \delta'$ si et seulement si $\mathcal{U}(\sigma, e) = (\sigma', i)$ et $c' = \kappa(\sigma, i)$.

Remarque 27. *Dans la définition de TAR, un état est associé à une liste de transitions mais si au début de l'algorithme la mémoire initiale est telle que la destination de la première transition est l'état initial, alors durant la construction l'état correspondra toujours à la destination de la transition la plus à gauche de la mémoire.*



$((\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge \mathsf{Fin}(2)) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(3) \wedge \mathsf{Inf}(4))$

(a) Automate portant plus de couleurs que d'arêtes



$(\mathsf{Inf}(5) \wedge \mathsf{Fin}(6) \wedge \mathsf{Fin}(7) \wedge \mathsf{Fin}(8)) \vee (\mathsf{Fin}(5) \wedge \mathsf{Inf}(7) \wedge \mathsf{Inf}(8))$

(b) Recoloriage de l'automate de la figure 5.5a avec autant de couleurs que d'arêtes

FIGURE 5.4. – Recoloriage d'un automate portant plus de couleurs que d'arêtes

Remarque 28. *En pratique cet algorithme ne devrait pas être utilisé. En effet, s'il existe plus de couleurs que d'arêtes, alors il suffit d'attribuer à chaque arête une couleur et de définir une condition décrivant les ensembles d'arêtes (et donc couleurs) qui doivent être vues infiniment souvent.*

Considérons par exemple l'automate de la figure 5.5a. On peut voir que la partie $(\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge \mathsf{Fin}(2)$ *indique qu'une exécution qui passe infiniment souvent par l'arête #1 mais finiment souvent par les autres est acceptante alors que la partie* $\mathsf{Fin}(1) \wedge \mathsf{Inf}(3) \wedge \mathsf{Inf}(4)$ *indique qu'une exécution passant finiment souvent par l'arête #1 et infiniment souvent par les arêtes #3 et #4 est acceptante.*

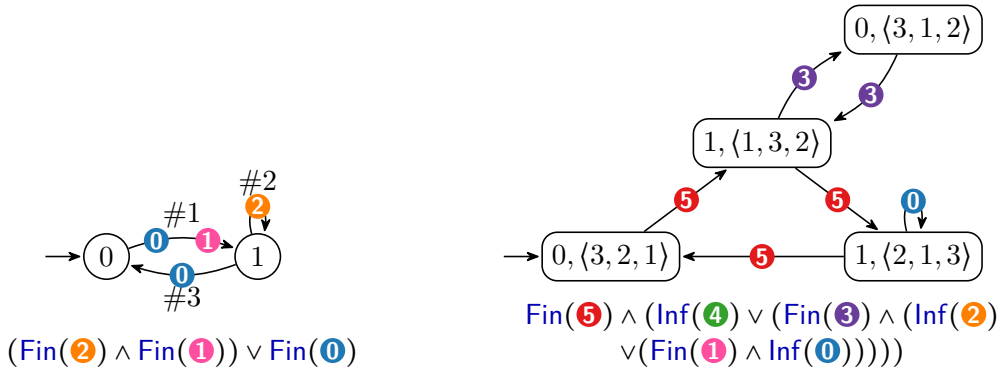
On introduit alors une couleur pour chaque arête (5 pour #1 par exemple) et décrit dans la condition quelles couleurs d'arêtes doivent être vues infiniment souvent. Ainsi voir infiniment souvent #1 et finiment souvent les autres peut être exprimé par $\mathsf{Inf}(5) \wedge \mathsf{Fin}(6) \wedge \mathsf{Fin}(7) \wedge \mathsf{Fin}(8)$.

Avec cette procédure nous obtenons alors un automate avec autant de couleurs que d'arêtes.

Remarque 29. *Puisqu'un état de l'automate* $\mathcal{A}^{TAR}$ *associe un des états de* $\mathcal{A}$ *à un des* $|E|!$ *ordres possibles sur les arêtes de* $\mathcal{A}$*, alors* $\mathcal{A}^{TAR}$ *possède au plus* $|Q| \times |E|!$ *états mais l'utilisation de la remarque 27 permet de limiter à* $|E|!$ *états.*

Exemple 11 (Application de TAR)**.** *Décrivons comment est obtenu l'automate de parité de la figure 5.5b à partir de l'automate de la figure 5.5a.*

La première étape est de créer une mémoire initiale portant les identifiants des arêtes. Ici $\langle 3, 2, 1 \rangle$ *a été choisi pour limiter la taille du résultat (voir section 5.11.4) mais n'importe quel ordre aurait été correct.*



(a) Automate à paritiser. Les étiquettes des arêtes sont leur identifiant.

(b) Résultat de l'application de TAR sur l'automate de la figure 5.5a

FIGURE 5.5. – Exemple d'application de TAR

Prendre l'arête #1 *depuis l'état 0 implique de déplacer 1 à gauche de l'historique, ce qui conduit à la création de l'historique* $\langle 1, 3, 2\rangle$. *Puisque l'ensemble* $\{0, 1\}$ *est rejetant et que 1 se trouvait en troisième position, la couleur émise est* $2 \times 3 + 1 = 5$.

5.5. State Appearance Record

Étudions maintenant l'algorithme *State Appearance Record* (SAR) tel que décrit par LÖDING [47].

Il prend en entrée un automate de Muller et la condition est donc décrite par des ensembles d'états qui doivent être vus infiniment souvent. À la différence des autres algorithmes que nous décrivons, le SAR de LÖDING produit un automate de parité sur les états.

L'idée est ici d'avoir un historique décrivant quels états ont été vus infiniment souvent. De manière analogue à CAR lorsqu'un état est vu lors d'une exécution, il est déplacé à gauche de l'historique.

Remarque 30. *À l'inverse de CAR, il peut être possible de trouver l'état courant à partir de l'historique. En effet, si lors de la construction le premier état construit porte* q_0 *à gauche, l'état de l'automate de départ associé à l'état courant du résultat correspond à l'état le plus à gauche de l'historique. C'est pour cela que nous n'associerons pas d'état à un historique dans l'automate résultant.*

Décrivons alors de manière formelle cette transformation d'un automate de Muller $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ en un automate de parité $\mathcal{A}^{\mathrm{SAR}}$.

Dans cette section le terme *historique* désignera un ordre total sur les éléments de Q et l'ensemble de ces ordres totaux sera noté $\Pi(Q)$.

Algorithme 4 : Algorithme de construction CAR

Entrée : Un automate $\mathcal{A}$ avec une condition α sur les états de $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$, un historique initial m_0
Sortie : Un automate de parité $\mathcal{P}^{\text{TAR}}$ équivalent à $\mathcal{A}$

```
   /* Création de l'état initial du résultat                          */
1  Q' ← {(q_0, m_0)}
2  δ' ← {}
3  todo ← {(q_0, m_0)}
4  tant que todo n'est pas vide faire
5      (q, m) ← todo.top()
6      todo.pop()
7      pour chaque arête e = (q, ℓ, c, q') partant de q faire
8          edge_pos ← m^{-1}(e)
9          m' ← edge_pos · (m ∖ e)
           /* Calcul de la couleur. ⊭ α vaut 1 si l'ensemble des
              couleurs de associé vérifie α                            */
10         c' ← 2 × edge_pos + (m(0), ..., m(edge_pos)) ⊭ α
11         si (q', m') ∉ Q' alors
12             todo.append((q', m'))
13             Q' ← Q' ∪ {(q', m')}
14         δ' ← δ' ∪ {((q, m), ℓ, c', (q', m'))}
15 retourner (Q', {0, ..., 2 × |M| + 1, Σ, δ', (q_0, m_0), parité maximale paire})
```

Le déplacement d'un état à gauche d'un historique s'appuiera sur une fonction $\mathcal{U} : \Pi(Q) \times Q \to \Pi(M) \times \mathbb{N}$ indiquant à quelle position se trouvait l'état déplacé qui est définie par :

$$\mathcal{U}(\sigma, q) = (\sigma', i)$$

où $\sigma' = \langle q \rangle \cdot \pi$ et π correspond à σ privé de q alors que $i = \sigma^{-1}(q)$.

Puisque SAR n'associe pas de couleurs aux arêtes ou aux états du résultat, la fonction κ de CAR et TAR n'a pas d'équivalent ici.

L'automate SAR associé à $\mathcal{A}$ est alors défini de la manière suivante :

Définition 61 (Automate SAR)**.** *Pour un automate de Muller $\mathcal{A} = (Q, \Sigma, \delta, q_0, T)$ tel que décrit précédemment, on définit l'automate SAR $\mathcal{A}^{SAR} = (Q^{SAR}, \delta^{SAR}, q_0^{SAR}, \alpha^{SAR})$ comme :*

- *$Q^{SAR} \subseteq \Pi(Q) \times \mathbb{N}$;*
- *q_0^{SAR} est un état dont l'élément le plus à gauche de l'historique est q_0 ;*
- *α^{SAR} est une condition de parité ;*

et $\delta^{SAR} \subseteq Q^{SAR} \times \Sigma \times Q^{SAR}$ est telle que

$$\delta^{SAR} = \left\{ (\pi, i) \xrightarrow{\ell} (\pi', j) \;\middle|\; \pi(0) \xrightarrow{\ell} \pi'(0) \in \delta, \mathcal{U}(\pi(0), \pi'(0)) = (\pi', j) \right\}$$

Il reste alors à définir la condition associée à $\mathcal{A}^{\mathrm{SAR}}$. Pour cela il faut adapter les notions de visualisation et séparation utilisées pour CAR.

Définition 62 (Historique affichant et séparant un ensemble, [47]). *Soit* $(\pi, h) \in Q^{SAR}$ *et* $F \subseteq Q$ *tel que* $|F| = m \geq 1$. *On dit que* (π, h) visualise F *si et seulement si* $\{\pi(0), \ldots, \pi(m-1)\} = F$.

On dit que (π, h) sépare F *si et seulement si* (π, h) *visualise* F *et* $h = m$.

On peut alors introduire la condition α' associée à $\mathcal{A}^{\mathrm{SAR}}$. Il s'agit d'une condition de la forme $\{(E_1, F_1), \ldots, (E_n, F_n)\}$ telle que

$$F_i = \{q \in Q^{\mathrm{SAR}} \mid q \text{ sépare un } F \subseteq Q \text{ avec } |F| \geq n - i + 1\}$$
$$E_i = F_i \setminus \{q \in Q^{\mathrm{SAR}} \mid q \text{ sépare un } F \in \mathcal{F} \text{ avec } |F| = n\}$$

Remarque 31. *Puisque* Q^{SAR} *est un sous-ensemble de* $\Pi(Q)$, $\mathcal{A}^{SAR}$ *a au plus* $|Q|!$ *états.*

5.6. Index Appearance Record

Alors que CAR peut aussi être utilisé pour transformer un automate de Rabin ou de Streett en automate de parité, nous allons maintenant décrire un algorithme plus adapté à ces classes de TELA.

Puisque les conditions de Streett correspondent au dual des conditions de Rabin, nous n'allons décrire cet algorithme que pour ces dernières.

Considérons un automate de Rabin $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ où $\alpha = \bigvee_{i \in \mathcal{I}} (\mathsf{Fin}(p_i) \wedge \mathsf{Inf}(r_i))$ et cherchons à créer un automate de parité maximale impaire $\mathcal{A}^{\mathrm{IAR}}$ qui lui est équivalent.

L'idée derrière *Index Appearance Record* (IAR) est de mémoriser les paires satisfaites durant une exécution. L'*historique* sera alors un ordre total sur l'ensemble $\mathcal{I}$ des indices des paires de α qui sera noté $\Pi(\mathcal{I})$ alors que Seq associe à un sous-ensemble de $\mathcal{I}$ un ordre total.

La fonction de mise à jour de l'historique $\mathcal{U} : \Pi(\mathcal{I}) \times \mathsf{Seq}(\mathcal{I}) \to \mathsf{Seq}(\mathcal{I})$ déplace des éléments à gauche de l'historique. À la différence de CAR, la fonction n'a pas besoin d'indiquer le nombre d'éléments déplacés.

Pour un ensemble C de couleurs il nous faut également introduire $\mathcal{M}(\sigma, C) = \max\big(\{-1\} \cup \big\{i \in \{0, \ldots, |\mathcal{I}| - 1\} \mid p_{\sigma(i)} \in C \vee r_{\sigma(i)} \in C\big\}\big)$, la plus grande position dans un historique σ d'un indice associé à une paire portant une couleur de C ou -1 si une telle paire n'existe pas.

De manière intuitive les indices des paires vont être déplacées de manière à ce que les couleurs interdites des paires dont l'indice est déplacé à gauche de l'historique correspond aux couleurs interdites vues pendant une exécution.

De manière plus formelle, la fonction de mise à jour $\mathcal{U}$ est définie de manière récursive par

$$\mathcal{U}(\sigma, ()) = \sigma$$
$$\mathcal{U}(\sigma, \{i\} \cup I) = \mathcal{U}(\langle i, \sigma(0), \sigma(1), \ldots, \sigma(j-1), \sigma(j+1), \ldots, \sigma(|\mathcal{I}-1|)\rangle, I) \text{ où } j = \sigma^{-1}(i)$$

Posons $i = \max(\{-1\} \cup \{i \in \{0,\dots,|\mathcal{I}|-1\} \mid p_{\sigma(i)} \in C\})$ le plus grand indice d'une paire dont la couleur interdite est un élément de C. Notons ensuite $m = \mathcal{M}(\sigma, C)$. Par définition $m \geq i$. Si $m > i$, alors $\pi_{\sigma(m)} \notin C$ et la paire de Rabin dont l'indice est $\sigma(m)$ est acceptante pour la transition courante au moins puisque nous avons vu sa couleur requise en évitant sa couleur interdite.

On peut alors introduire une fonction $\kappa : \Pi(\mathcal{I}) \times 2^M \to \mathcal{I}'$ où $\mathcal{I}' = \{0, 1, \dots, 2 \times |\mathcal{I}|\}$.

$$\kappa(\sigma, C) = \begin{cases} 0 & \text{si } m = -1 \\ 2m+1 & \text{si } p_{\sigma(m)} \notin C \\ 2m+2 & \text{si } p_{\sigma(m)} \in C \end{cases}$$

où $m = \mathcal{M}(\sigma, C)$.

De la même manière que pour CAR l'ordre de déplacement des indices dans l'historique n'influe pas sur la correction du résultat. Cependant pour préserver le déterminisme de la construction, on doit définir une fonction de choix d'ordre d'insertion. De manière formelle un *choix d'ordonnancement de Rabin* pour un TELA $\mathcal{A}$ est une fonction $f : \Pi(\mathcal{I}) \times \delta \to \mathsf{Seq}(\mathcal{I})$ telle que $f(\sigma, q \xrightarrow{\ell,C} q')$ est un ordonnancement de l'ensemble $\{i \in \{0,\dots,|\mathcal{I}|-1\} \mid p_{\sigma(i)} \in C\}$ des paires de Rabin dont la couleur interdite est un élément de C.

On peut alors définir l'automate issu de la construction IAR.

Définition 63 (Automate IAR)**.** *Soit $\sigma_0 \in \Pi(\mathcal{I})$ et f un choix d'ordonnancement de Rabin sur $\mathcal{A}$. On définit le TELA $\mathcal{A}^{IAR} = (Q^{IAR}, \mathcal{I}^{IAR}, \Sigma, \delta^{IAR}, (q_0, \sigma_0), \alpha^{IAR})$*

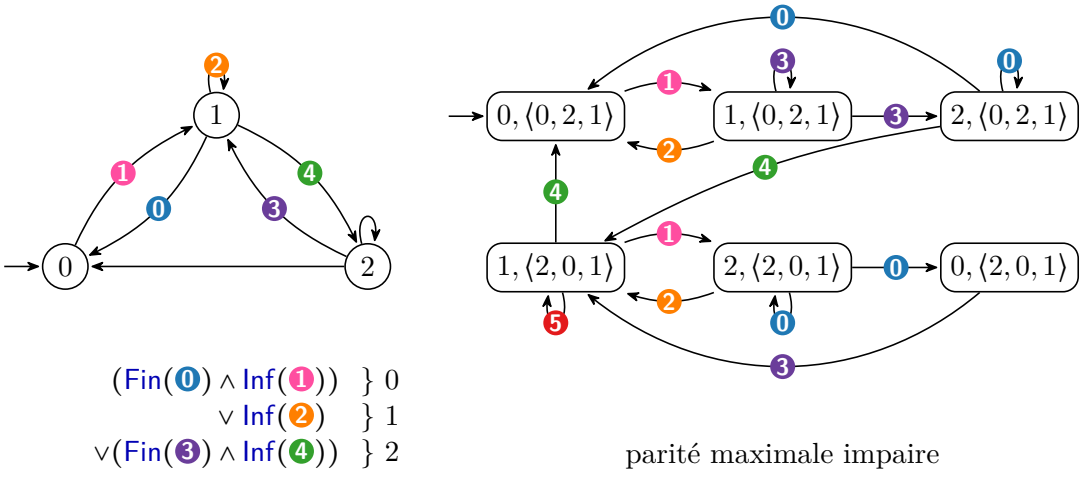
- *$Q^{IAR} \subseteq Q \times \Pi(\mathcal{I})$;*
- *$\mathcal{I}^{IAR} = \{0, \dots, 2 \times |\mathcal{I}|\}$;*
- *α^{IAR} est une condition de parité maximale impaire ;*

et δ^{IAR} est telle que pour tout $d = (q \xrightarrow{\ell,C} q') \in \delta$ et pour tout $\sigma \in \Pi(\mathcal{I})$, $d' = (q, \sigma) \xrightarrow{\ell,C'} (q', \sigma')$ appartient à δ' où $\sigma' = \mathcal{U}(\sigma, f(\sigma, d))$ et $C' = \{\kappa(\sigma', C)\}$.

Exemple 12 (Construction IAR)**.** *Considérons l'automate de la figure 5.6a et montrons comment a été construit le DPA de la figure 5.6b. Puisqu'il s'agit d'une condition Rabin-like à 3 paires, un historique porte les indices de 0 à 2. De manière arbitraire on décide d'utiliser $\langle 0, 2, 1\rangle$ comme historique initial. Depuis l'état $(2, \langle 0, 2, 1\rangle)$, l'état $(1, \langle 2, 0, 1\rangle)$ est créé puisque dans l'automate de départ, ❸ est vue. Cette couleur est la couleur interdite de la paire 2. L'indice 2 est donc déplacé à gauche de l'historique. Puisque cet indice était à la position 1 et que ❸ est rejetant, la couleur ❹ $= 2 \times 1 + 2$ est émise.*

À l'inverse, depuis l'état $(0, \langle 2, 0, 1\rangle)$, le passage vers l'état 1 dans l'automate de départ implique de voir ❶ qui est la couleur requise de la paire 0. Cette dernière n'est donc pas déplacée à gauche de l'historique. De plus la couleur ❷ $= 2 \times 0 + 2$ est émise car 0 est à la position 1 et ❸ est acceptant.

Remarque 32. *Comme pour CAR, la version de IAR implémentée dans Spot diffère de l'algorithme présenté ici par un décalage de -1 sur les couleurs produites et donc l'adoption*



(a) Automate Rabin-like à paritiser (b) Automate de parité obtenu en appliquant IAR sur l'automate de la figure 5.6a

FIGURE 5.6. – Exemple de construction IAR sur un automate Rabin-like

d'une condition de parité maximale paire pour les automates de Rabin. Cela permet de réduire l'ensemble $\mathcal{I}^{IAR}$ des couleurs du DPA de sortie à $\{0, \ldots, 2 \times |\mathcal{I}| - 1\}$, ce qui peut être important puisque Spot impose une limite sur la couleur maximale (31 par défaut).

Théorème 7 (Correction de IAR, [47]). *On a $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{IAR})$. De plus, si $\mathcal{A}$ est déterministe, alors $\mathcal{A}^{IAR}$ l'est également.*

Remarque 33. *Puisque $Q^{IAR} \subseteq Q \times \Pi(\mathcal{I})$, la procédure crée un automate avec au plus $|Q| \times |\mathcal{I}|!$ états dans le pire cas. Cependant, sauf si des couleurs apparaissent dans beaucoup de paires, on a en général $|\mathcal{I}| \leq \frac{|M|}{2}$, ce qui fait que IAR est souvent théoriquement un meilleur choix que CAR lorsqu'il est possible de l'utiliser.*

5.7. Dégénéralisation

Nous parlons maintenant d'une méthode nommée *dégénéralisation* [4] permettant de transformer un automate avec une condition de Büchi généralisée en automate avec une condition de Büchi. Comme pour LAR cela va passer par la création de copies des états. Même s'il s'agit une nouvelle fois d'associer à chaque copie d'un état de la mémoire, nous utiliserons le terme de niveau. Avant d'évoquer la transformation dans le cas où les transitions portent les couleurs, décrivons le cas des automates pour lesquels ce sont les états qui sont colorés. Plaçons nous dans le cadre de la création d'un automate de Büchi $\mathcal{B}$ à partir d'un automate de Büchi généralisé $\mathcal{A}$ dont la condition porte sur un ensemble M de m couleurs. La première chose à faire est de créer un ensemble L de $m+1$ copies (niveaux) de $\mathcal{A}$ (ce qui donnera un automate avec au plus $n(m+1)$ états). Le principe est ensuite d'ordonner les éléments de M et de passer du niveau i au niveau $i+1$ lorsque

la i^{e} couleur est vue. Les états du dernier niveau sont tous acceptants et toute transition partant d'un état du dernier niveau va au premier niveau.

Avec cette construction, on est obligé de voir toutes les couleurs infiniment souvent pour visiter infiniment souvent le dernier niveau, d'où la correction de la construction.

Considérons maintenant le cas des TELA. Dans ce cas, il suffit de m niveaux et de colorer une transition quittant le niveau m pour le premier.

De manière plus formelle définissons la fonction de passage $\mathcal{S} : L \times 2^M \to L \times \mathbb{B}$ prenant en entrée un niveau i, un ensemble de couleurs C et retournant un nouveau niveau tout en indiquant si la nouvelle transition doit être colorée.

$$\mathcal{S}(i, C) = \begin{cases} (j, \bot) & \text{si } j < |D| \\ (j - |D|, \top) & \text{si } j \geq |D| \end{cases}$$

où $j = \max(k \in \{i, \dots, i + |D|\} \mid \{d_{i \mod |D|}, \dots, d_{(k+|D|-1) \mod |D|}\} \subseteq C)$ est la taille de la plus longue séquence d'éléments consécutifs de D commençant par d_i qui peut être trouvée dans C [30, 4].

Ici la définition de $\mathcal{S}$ ajoute une subtilité. Elle indique qu'au lieu de retourner au premier niveau, elle va aller au niveau $j - |D|$. Cela signifie que si des couleurs qui sont présentes sur la transition permettent de passer des niveaux depuis le premier niveau, alors on va continuer à avancer.

On peut alors définir l'automate de Büchi issu de la dégénéralisation ainsi :

Définition 64. *Soit $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ un automate de Büchi généralisé. L'automate de Büchi $\mathcal{B} = (Q', \{0\}, \Sigma, \delta', q_0', Büchi)$ obtenu à l'aide de la dégénéralisation est tel que :*

- *$Q' \subseteq Q \times |M|$;*
- *$q_0' = (q_0, l)$ pour un $l \in \{0, \dots, |M| - 1\}$;*

et δ' est telle que

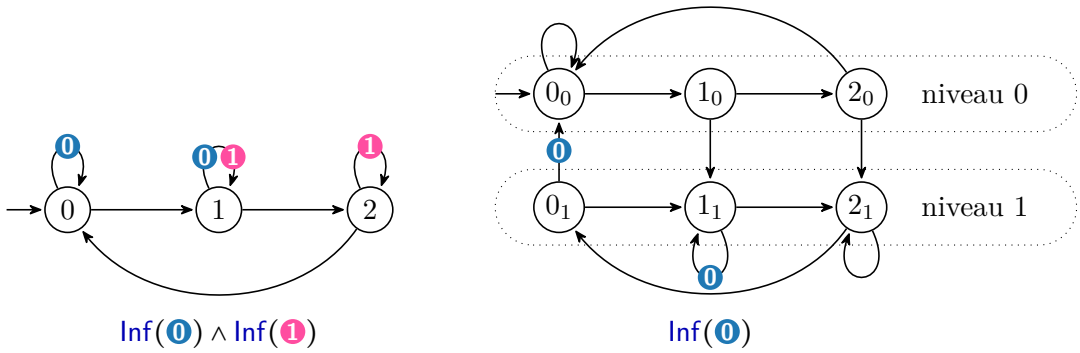
- *$(q, l) \xrightarrow{\ell, 0} (q', l') \in \delta'$ si et seulement si $q \xrightarrow{\ell, c} q' \in \delta'$ et $\mathcal{S}(l, c) = (l', \top)$;*
- *$(q, l) \xrightarrow{\ell, \varnothing} (q', l') \in \delta'$ si et seulement si $q \xrightarrow{\ell, c} q' \in \delta'$ et $\mathcal{S}(l, c) = (l', \bot)$.*

Théorème 8 (Correction de la dégénéralisation, [4])**.** *Soit $\mathcal{A}$ un automate de Büchi généralisé et $\mathcal{B}$ le résultat de la dégénéralisation de $\mathcal{A}$. Alors $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. De plus, $\mathcal{A}$ est déterministe si et seulement si $\mathcal{B}$ l'est.*

Exemple 13 (Dégénéralisation d'un TGBA)**.** *Considérons l'automate de Büchi généralisé de la figure 5.7a. L'automate de Büchi de la figure 5.7b correspond au résultat de l'application de la dégénéralisation sur cet automate. Un état y est nommé q_i où q correspond au numéro de l'état de l'automate de départ alors que i correspond au niveau.*

La première étape a consisté à créer deux copies de l'automate puisqu'il y a deux couleurs dans l'automate de départ.

Ensuite il est décidé que pour passer du niveau 0 au niveau 1, il faut voir la couleur 1 alors que 0 permet de passer du niveau 1 au niveau 0.



(a) Automate de Büchi généralisé (b) Automate de Büchi équivalent

FIGURE 5.7. – Exemple de dégénéralisation d'un automate de Büchi généralisé

L'étape suivante correspond à la redirection des arêtes en fonction des couleurs qu'elles portent. Ainsi puisque la boucle de 2 *porte* 1 *alors on a une arête allant de* 2_0 *à* 2_1*. De même il existe une arête allant de* 0_1 *à* 0_0 *car* 0_1 *est au niveau 1 et l'arête associée dans l'automate de départ porte* 0*. Cette arête porte* 0 *puisqu'elle relie le dernier au premier niveau.*

Il nous reste à évoquer la boucle de l'état 1*. Puisque celle-ci porte plusieurs couleurs, on a le choix entre plusieurs niveaux. Ainsi depuis le niveau 0, on peut décider de relier* 1_0 *à* 1_1 *sans émettre de couleur (et donc oublier que l'on a vu* 1 *sur cette arête) ou créer une boucle sur* 1_0 *colorée. Il en est de même pour la boucle de l'état* 1_1*.*

5.8. Dégénéralisation partielle (Contribution)

La dégénéralisation est une méthode visant à transformer un automate portant une condition de Büchi généralisée en un automate à condition de Büchi par la création de multiples niveaux.

Cependant dans le cadre de la synthèse LTL, on peut être amené à devoir traiter des automates dont la condition est par exemple $\mathsf{Fin}(0) \vee (\mathsf{Inf}(1) \wedge \mathsf{Inf}(2))$. Même s'il ne s'agit pas d'une condition de Büchi généralisée, on voit que la partie $\mathsf{Inf}(1) \wedge \mathsf{Inf}(2)$ en est une. On va alors chercher à remplacer dans la formule cette conjonction de Inf par un unique Inf.

De manière plus formelle, étant donné un TELA $\mathcal{A}$ et un sous-ensemble D de ses couleurs, le but est de modifier $\mathcal{A}$ de manière à pouvoir remplacer toute sous-formule de la forme $\bigwedge_{d \in D} \mathsf{Inf}(d)$ (resp. $\bigvee_{d \in D} \mathsf{Fin}(d)$) par un unique $\mathsf{Inf}(e)$ (resp. $\mathsf{Fin}(e)$) où e est une couleur qui ne fait pas partie de $\mathcal{A}$. Une telle substitution pour une condition α sera notée $\alpha\left[\bigwedge_{d \in D} \mathsf{Inf}(d) \leftarrow \mathsf{Inf}(e)\right]\left[\bigvee_{d \in D} \mathsf{Fin}(d) \leftarrow \mathsf{Fin}(e)\right]$.

De manière analogue à la dégénéralisation, il s'agira de poser un ordre $(d_0, d_1, \ldots, d_k)$ sur les éléments de D et d'associer à chaque état du résultat un niveau dans $L = \{0, \ldots, |D| - 1\}$. Le passage d'un niveau i au niveau $i + 1$ se fera lorsqu'une arête portant

la couleur d_i est traitée et le i-ème niveau sera atteint après avoir vu les i premières couleurs de D. Après avoir vu la dernière couleur de D, un retour au premier niveau est effectué et une nouvelle couleur e est émise.

Remarque 34. *Avec la dégénéralisation, la couleur produite pouvait être* 0*. Ici, la production d'une nouvelle couleur nous permet de traiter les cas tels que* $(\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(0) \vee \mathsf{Fin}(1) \vee \mathsf{Fin}(2))$ *pour le sous-ensemble de couleurs* $\{0,1,2\}$*.*

En effet, par définition, puisque $\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)$ *ne porte pas la couleur* 2*, cette conjonction ne sera pas remplacée par un unique* Inf*. Il faudra dans le résultat que* 0 *et* 1 *soient toujours présents pour détecter cette conjonction alors que la nouvelle couleur sera associée au nouveau* Fin*.*

Le théorème suivant définit l'automate résultant de la dégénéralisation partielle et affirme la correction du résultat.

Définition 65 (Correction de la dégénéralisation partielle)**.** *Soit* $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ *un TELA,* $D = \{d_0, d_1, \dots, d_{|D|-1}\}$ *un ensemble ordonné de couleurs de* M *et soit* $L = \{0, 1, \dots, |D|-1\}$ *un ensemble de niveaux.*

Soit $\mathcal{A}' = (Q', M', \Sigma, \delta', (q_0, i_0), \alpha')$ *le résultat de la dégénéralisation partielle de* $\mathcal{A}$ *selon* D *où*

- i_0 *peut être n'importe quel élément de* L *;*
- $Q' = Q \times L$ *;*
- $M' = M \cup \{e\}$ *pour une couleur* $e \notin M$ *;*
- $\alpha' = \alpha\,[\bigwedge_{d \in D} \mathsf{Inf}(d) \leftarrow \mathsf{Inf}(e)]\,[\bigvee_{d \in D} \mathsf{Fin}(d) \leftarrow \mathsf{Fin}(e)]$ *;*
- $\delta' = \left\{(q_1, i) \xrightarrow{\ell, C} (q_2, j) \in Q' \times \Sigma \times M' \times Q' \;\middle|\; q_1 \xrightarrow{\ell, C \cap M} q_2 \in \delta,\ \mathcal{S}(i, C \cap M) = (j, C \setminus M)\right\}$
  où $\mathcal{S}$ *est définie comme :*

$$\mathcal{S}(i, C) = \begin{cases} (j, \varnothing) & \text{si } j < |D| \\ (j - |D|, \{e\}) & \text{si } j \geq |D| \end{cases}$$

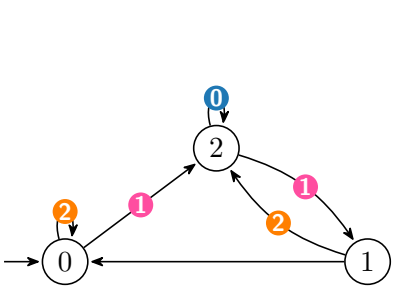
où $j = \max(k \in \{i, \dots, i + |D|\} \mid \{d_{i \bmod |D|}, \dots, d_{(k+|D|-1) \bmod |D|}\} \subseteq C)$ *est la taille de la plus longue séquence de couleurs consécutives de* D *à partir de* d_i *qui peut être trouvée dans* C*.*

Exemple 14. *Considérons l'exemple de la figure 5.8 où on cherche à dégénéraliser l'ensemble* $\{1,2\}$*. On y considère que pour passer du niveau 0 au niveau 1, il faut voir la couleur* 2 *alors que le passage du niveau 1 au niveau 0 se fait en voyant la couleur* 1*.*

Les copies de l'état 2 doivent conserver la boucle portant 0 *puisqu'elle ne fait pas partie de l'ensemble traité.*

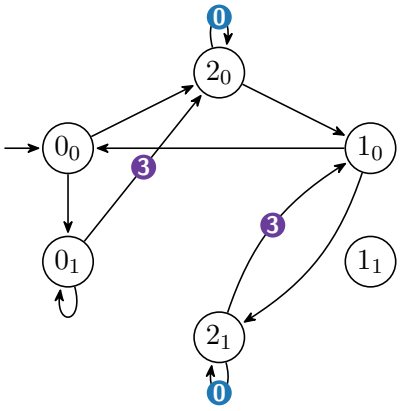
Le reste de la construction se fait de manière analogue à la dégénéralisation classique.

Remarque 35. *La dégénéralisation partielle n'impose aucune contrainte sur* D*. Ainsi avec une condition telle que* $\mathsf{Fin}(0) \vee \mathsf{Inf}(1)$ *et* $D = \{0,1\}$*, deux niveaux ainsi qu'une couleur* 2 *(sur l'automate, pas dans la condition) seront créés alors que la condition ne changera pas.*



$\mathsf{Fin}(0) \vee (\mathsf{Inf}(1) \wedge \mathsf{Inf}(2))$

(a) Automate dont la condition porte une conjonction de Inf



$\mathsf{Fin}(0) \vee \mathsf{Inf}(3)$

(b) Résultat de la dégénéralisation partielle pour l'ensemble $\{1, 2\}$ de l'automate de la figure 5.8a

FIGURE 5.8. – Exemple de dégénéralisation partielle

Remarque 36. *Si cette procédure est appliquée sur un automate de Büchi généralisé (resp. co-Büchi généralisé), alors après le remplacement de la nouvelle couleur e par* 0 *(qui est forcément possible), le résultat est un automate de Büchi (resp. co-Büchi). La dégénéralisation partielle est donc une extension de la dégénéralisation.*

Remarque 37. *La transformation d'un automate portant condition de Rabin généralisée en un automate portant une condition de Rabin peut être faite en appliquant la dégénéralisation partielle. En effet, une condition de Rabin généralisée est de la forme* $\bigvee_{i\in\mathcal{I}}(\mathsf{Fin}(m_i) \wedge \mathsf{Inf}(m_i^1) \wedge \mathsf{Inf}(m_i^2) \wedge \ldots \wedge \mathsf{Inf}(m_i^k))$ *et il suffit d'appliquer* $|\mathcal{I}|$ *dégénéralisations partielles sur les ensembles* $\{m_i^1, m_i^2, \ldots, m_i^k\}_{i\in\mathcal{I}}$ *pour obtenir un automate de Rabin.*

5.9. Automates Büchi-type

Nous décrivons ici une transformation permettant, lorsque c'est possible, de recolorier les arêtes d'un automate et de lui assigner une condition de Büchi sans en changer le langage.

Les automates pour lesquels une telle transformation existe sont qualifiés de Büchi-type.

Définition 66 (Automate Büchi-type)**.** *Un automate* $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ *est dit Büchi-type s'il existe une fonction de coloration* $f : \delta \to 2^{\{0\}}$ *telle que l'automate de Büchi* $\mathcal{A}' = (Q, \{0\}, \{q, \xrightarrow{\ell, f(c)} q' \mid q \xrightarrow{\ell, c} q' \in \delta\}, q_0, \mathsf{Inf}(0))$ *lui est équivalent.*

De manière générale cette notion peut être étendue à tout type de condition. Un automate est X-type s'il est possible de le "recolorier" en un automate équivalent de condition X.

Krishnan et al. [44] décrivirent des moyens de construire de tels automates mais pas dans le cadre des TELA. Cette construction est faite en deux étapes. La première consiste à produire une fonction colorant une arête si elle n'est dans aucun cycle rejetant. Cette méthode permet d'obtenir un automate reconnaissant un langage au plus égal à celui de départ. Pour vérifier si l'automate est équivalent à l'automate original, un test d'équivalence est fait.

Même si le test d'équivalence est en général complexe, le premier point à mettre en avant est que l'on sait que l'automate construit reconnaît un sous-ensemble du langage de l'automate de départ et il n'y a donc qu'un test d'inclusion à faire.

Ensuite puisque les deux automates partagent la même structure de transition, ce test se résume à retirer les arêtes que nous avons colorées et de regarder s'il reste un cycle acceptant.

Nous allons ici montrer l'idée derrière la méthode telle que présentée par Krishnan, Puri et Brayton pour les automates de Muller avant de décrire le cas des conditions de Rabin. Nous allons ensuite montrer l'adaptation de cet algorithme au cadre des TELA qui était déjà présente dans Spot pour les conditions de Rabin.

5.9.1. Cas des automates de Muller

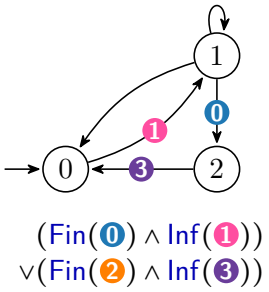
Considérons un automate de Muller $\mathcal{A} = (Q, \Sigma, \delta, q_0, T)$. Notre but est de trouver un automate de Büchi équivalent portant la même structure de transitions. Le principe derrière l'algorithme de Krishnan et al. et de construire une condition de Büchi T' telle que $\mathcal{B} = (Q, \Sigma, \delta, q_0, T')$ reconnaît le plus grand langage tel que $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$. Comme évoqué plus tôt, on cherche alors à savoir si $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{L}(\mathcal{A})$, ce qui est relativement simple puisque les deux automates partagent la même structure de transitions.

Le point important est donc de trouver si un état doit être acceptant (un élément de T') ou non. Pour cela, l'idée est que dans un automate de Büchi, si un état est acceptant, alors tout cycle passant par cet état sera acceptant. Ainsi, pour un état donné de l'automate de départ, s'il existe un cycle rejetant passant par cet état, alors il ne pourra pas être acceptant dans l'automate résultant.

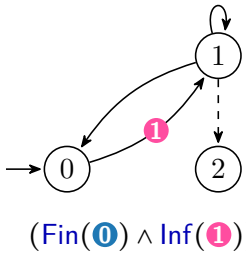
Dans le cas général, calculer si un état doit être coloré ou non est complexe, puisque cela implique la résolution du problème de vacuité [6]. Cependant dans le cas des automates de Rabin, la structure particulière de la condition permet une recherche plus efficace. Pour rappel (définition 26), dans le cadre de l'acceptation basée sur les états, une condition de Rabin est de la forme $\{(E_1, F_1), \dots (E_n, F_n)\}$.

Remarque 38. *La définition de condition de Rabin utilisée par Krishnan et al. diffère de la nôtre. Une paire (E_i, F_i) y est acceptante si au moins un état visité infiniment souvent est dans E_i et que l'ensemble des états visités infiniment souvent est inclus dans F_i. Pour passer de cette définition à la nôtre, il suffit de complémenter F_i.*

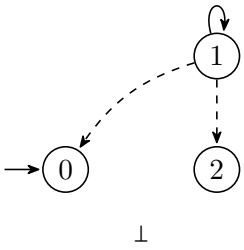
Pour déterminer si un état s est acceptant, on commence par calculer $\mathcal{S}$ l'ensemble



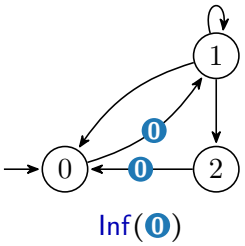
(a) Automate de Rabin



(b) Résultat de la première étape



(c) Résultat de la deuxième étape



(d) Automate de Büchi équivalent à celui de la figure 5.9a

FIGURE 5.9. – Exemple de traitement d'un automate Büchi-type

des états de la SCC contenant s. Étant donnée une paire (E_i, F_i) telle que $\mathcal{S} \cap \mathsf{F}_i = \varnothing$, on supprime les états de E_i. Ces derniers ne sont dans aucun cycle rejetant. On applique ensuite la même procédure sur le sous-graphe obtenu.

5.9.2. Cas des TELA

Nous allons maintenant voir comment cette méthode a été modifiée pour s'adapter aux TELA. Spot ne pouvait traiter que les automates de Rabin et nous allons donc nous limiter à ces conditions.

L'idée est la même sauf qu'au lieu de supprimer des états de l'automate, il s'agit des arêtes. Nous ne traiterons que les arêtes ne reliant pas deux SCC puisque leur coloration n'a pas d'impact sur l'acceptation. Pour chaque paire telle que sa couleur interdite n'est pas dans la SCC courante, toute arête portant la couleur requise sera toujours acceptante et portera donc 0 dans le résultat.

Remarque 39. *Cette analyse ne porte que sur la structure de l'automate. Ainsi, même si* KRISHNAN, PURI *et* BRAYTON *ne s'appuient que sur des automates déterministes, il est possible d'appliquer la même procédure sur un automate non-déterministe. Si un résultat est trouvé, alors il est correct mais un automate Büchi-type non-déterministe peut ne pas être détecté comme tel.*

Exemple 15 (Recoloration d'un automate Büchi-type)**.** *Illustrons cette procédure sur l'automate de la figure 5.9a.*

On y voit un automate composé d'une seule SCC dans laquelle 2 n'apparaît pas. Puisque la paire (2,3) porte 2 comme couleur interdite, alors toute arête portant la couleur 3 ne sera dans aucun cycle rejetant.

Cette arête est alors masquée et nous avons alors (figure 5.9b) deux SCC. Celle liée à l'état 2 ne porte pas d'arête et ne sera donc pas traitée. L'arête en pointillés relie deux SCC et on choisit de ne pas lui attribuer de couleur dans le résultat.

Dans cet automate, la couleur 1 n'est pas présente. On peut alors rechercher toutes les arêtes portant 0 et indiquer qu'elles devront être acceptantes dans le résultat.

Lorsque ces arêtes sont masquées, on obtient l'automate de la figure 5.9c. Puisque la condition d'acceptation associée est $\bot$, la seule arête encore présente ne pourra pas être coloriée.

Après recoloration des arêtes on obtient alors l'automate de la figure 5.9d qui est équivalent à celui de la figure 5.9a.

5.10. Automates parity-type (Contribution)

Les automates *parity-type* sont une généralisation des automates Büchi-types. Au lieu de poser une condition de Büchi sur un automate sans en changer la structure, les arêtes sont maintenant recolorées de manière à attribuer une condition de parité.

Pour comprendre l'idée derrière cette construction, appuyons-nous sur l'automate de parité de la figure 5.10a.

Il s'agit dans un premier temps de détecter les arêtes qui ne sont dans aucun cycle acceptant, comme pour la construction des automates Büchi-type. Dans l'automate de la figure 5.10a, il s'agit des arêtes colorées par 1 et celle reliant l'état 3 à l'état 0.

Supprimer ces arêtes conduit à l'obtention de plusieurs SCC. On répète alors le processus mais pour rechercher les arêtes qui ne sont dans aucun cycle rejetant. Il s'agit des arêtes portant 0. Après les avoir supprimés, il n'y a plus d'arête dans l'automate.

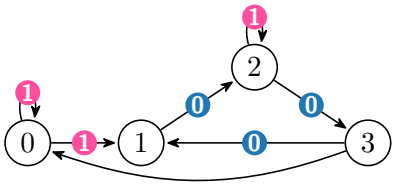
La dernière étape consiste à recolorier l'automate. Puisque nous avons commencé par supprimer des arêtes toujours rejetantes, les premières arêtes supprimées auront la plus grande couleur impaire k. Les arêtes supprimées à l'étape suivante auront la couleur paire $k-1$.

L'automate résultant a alors une condition de parité maximale paire et est décrit dans la figure 5.10c.

Remarque 40. *À la différence du cas des automates Büchi-type, il n'est pas nécessaire d'effectuer un test d'inclusion du résultat. En effet le résultat est correct si et seulement si toutes les arêtes ont été colorées.*

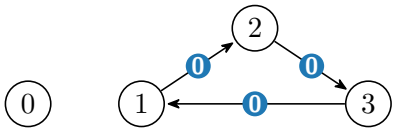
5.11. Combinaison des transformations existantes (Contribution)

Cette section vise à combiner les différentes techniques qui ont été évoquées jusqu'ici. Ce sera également l'occasion de décrire différentes heuristiques qui sont utilisées. L'algo-



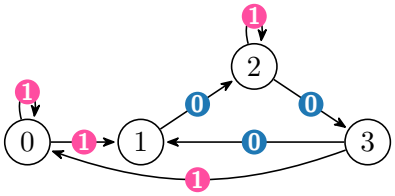
Fin(1) ∧ Inf(0)

(a) Automate de parité



Fin(1) ∧ Inf(0)

(b) Automate de parité de la figure 5.10a sans les arêtes portant 1



Fin(1) ∧ Inf(0)

(c) Automate de parité équivalent à celui de la figure 5.10a

FIGURE 5.10. – Exemple d'analyse des cycles d'un automate de parité

rithme général combinant ces procédures correspond à la fonction `to_parity` de Spot. Une très grande partie des optimisations et algorithmes qui lui sont associés peut être désactivée à l'aide d'options.

5.11.1. Découpage en SCC

Avant de décrire l'algorithme général, il nous faut introduire le principe du découpage en plusieurs composante fortement connexe. Chaque composante est paritisée indépendamment avant de les regrouper en un unique automate.

Nous verrons page 79 que cela permet notamment de tirer parti des propriétés plus locales d'un automate.

Les automates peuvent alors être recombinés. Par exemple puisque l'état 1 est relié à l'état 2 dans l'automate de départ, alors toutes les copies de 1 sont reliées à une copie de 2.

5.11.2. Algorithme général

Nous allons maintenant regrouper les différentes transformations qui ont été décrites jusqu'ici en un algorithme global que nous désignerons par *to_parity*.

Pour des questions de lisibilité, les paritisations récursives liées à l'utilisation des préfixes de parité par exemple ne seront pas incluses.

Comme nous avons indiqué page 28, TAR n'est pas censé être meilleur que CAR s'il y a une simplification de l'automate, nous n'allons donc pas l'utiliser ici tout comme SAR qui est adapté aux automates de Muller.

Puisque les automates parity-type sont une généralisation des automates Büchi-type, tester si un automate est Büchi-type ne sera pas fait.

Construction des sous-automates de parité

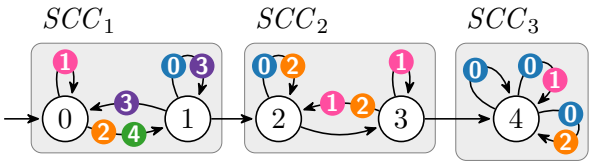
La première étape pour paritiser un TELA $\mathcal{A}$ consiste à énumérer les SCC de l'automate de départ. Il s'agira ensuite de construire des automates de parité pour chacun de ces sous-automates. En détail, cela signifie que l'on va énumérer les SCC S_i de $\mathcal{A}$ et pour chaque sous-automate $\mathcal{A}_{|S_i}$ associé à ces SCC, appliquer les étapes suivantes :

1. Propager les couleurs (voir page 87) ;
2. Appliquer les simplifications ;
3. Si l'automate porte une conjonction de Inf ou une disjonction de Fin, appliquer une dégénéralisation partielle. Puisque la condition associée à l'automate a changé, retourner à l'étape 1 ;
4. Transformer l'automate en un automate de parité maximale R_i en utilisant la première procédure applicable dans cette liste :
   - Si $\mathcal{L}(\mathcal{A}_{|S_i})$ est vide [6], supprimer toutes les couleurs et attribuer la condition d'acceptation $\bot$ (qui est un cas particulier de parité maximale paire),
   - Si la condition est déjà une condition de parité maximale, alors ne rien faire,
   - Si la condition d'acceptation a la forme $\mathsf{Inf}(m_0) \vee (\mathsf{Fin}(m_1) \wedge (\mathsf{Inf}(m_2) \vee \ldots))$ d'une condition de parité maximale, renuméroter les couleurs m_0, m_1, ... par ordre décroissant pour obtenir une condition de parité maximale,
   - Si l'automate est parity-type, appliquer la recoloration associée pour obtenir un automate de parité maximale,
   - Si l'automate est Rabin-like ou Streett-like et que la condition porte moins de paires que de couleurs, appliquer IAR,
   - Sinon, appliquer CAR.

Illustrons cette première étape sur l'automate de la figure 5.11 à l'aide de la figure 5.12.

Commençons par le traitement de la SCC 1. Puisque $\mathsf{Fin}(1) \vee \mathsf{Fin}(3)$ est présent dans la condition de cet automate, on applique une dégénéralisation partielle. Puisque la structure de l'automate a changé, on applique de nouveau une simplification et une propagation. On doit maintenant appliquer un des algorithmes de l'étape 4. La première étape qui peut être appliquée est l'application de CAR, ce qui conduit à l'obtention d'un automate de parité maximale paire à quatre états.

Traitons maintenant la SCC 2. Puisque seules les couleurs 0, 1 et 2 sont présentes, on peut restreindre la condition à ces couleurs et on obtient la condition de Rabin $(\mathsf{Inf}(2) \wedge \mathsf{Fin}(1)) \vee (\mathsf{Inf}(1) \wedge \mathsf{Fin}(0))$. Cette condition porte deux paires et trois couleurs.



$$\big(\mathsf{Inf}(2)\wedge\mathsf{Fin}(1)\big)\vee\big((\mathsf{Inf}(4)\vee\mathsf{Inf}(1))\wedge\mathsf{Fin}(0)\wedge(\mathsf{Fin}(1)\vee\mathsf{Fin}(3))\big)$$

FIGURE 5.11. – TELA à paritiser. Pour des questions de lisibilité, les étiquettes ne sont pas représentées.

On va donc appliquer IAR et obtenir un automate de parité maximale impaire à trois états.

Pour la dernière SCC, la simplification s'appuie sur la présence de 0 sur toutes les arêtes ainsi que sur l'absence de 3 et 4 pour réécrire la condition comme $\mathsf{Inf}(2)\wedge\mathsf{Fin}(1)$. Remplacer 2 par 0 dans cette condition nous permet d'obtenir un automate de parité maximale paire.

Fusion des SCC

La procédure générale a créée un ensemble d'automates de parité et il nous reste alors à relier ces SCC entre elles. Cependant, ils n'ont pas tous le même type d'acceptation. Par exemple, puisque CAR a été appliqué sur la SCC 1, alors un automate de parité maximale paire a été obtenu alors que IAR a été utilisé pour la deuxième SCC, ce qui a donné un automate de parité maximale impaire.

Pour regrouper ces automates, il faut alors harmoniser leur condition d'acceptation. Pour cela, on peut appliquer la même procédure que dans l'exemple 7.

Une fois cette harmonisation effectuée, il nous reste à ajouter les arêtes entre ces SCC. Puisqu'il y a dans l'automate de départ une arête entre l'état 1 et l'état 2, alors il devra y avoir une arête entre chaque copie de 1 et n'importe quelle copie de 2. Il en est de même pour l'arête entre les états 3 et 4.

L'automate obtenu est alors celui de la figure 5.13.

Minimisation du nombre de couleurs du résultat

Puisqu'il est généralement préférable de travailler sur des automates ayant peu de couleurs, une dernière étape consiste à minimiser le nombre de couleurs de l'automate résultant. Pour cela nous appliquons la méthode décrite par CARTON et MACEIRAS [13] permettant de calculer l'index de Rabin (c'est-à-dire le nombre minimal de paires nécessaire) d'un automate.

De manière globale, il s'agit de la même procédure que pour la création des automates parity-type (section 5.10). On va retirer les arêtes avec la plus grande couleur, ce qui va donner un ensemble de SCC. On va ensuite appliquer la même procédure sur les SCC résultantes jusqu'à la suppression de toutes les arêtes. Le coloriage de l'automate résultant se fait alors en fonction du moment où a été supprimé une arête.

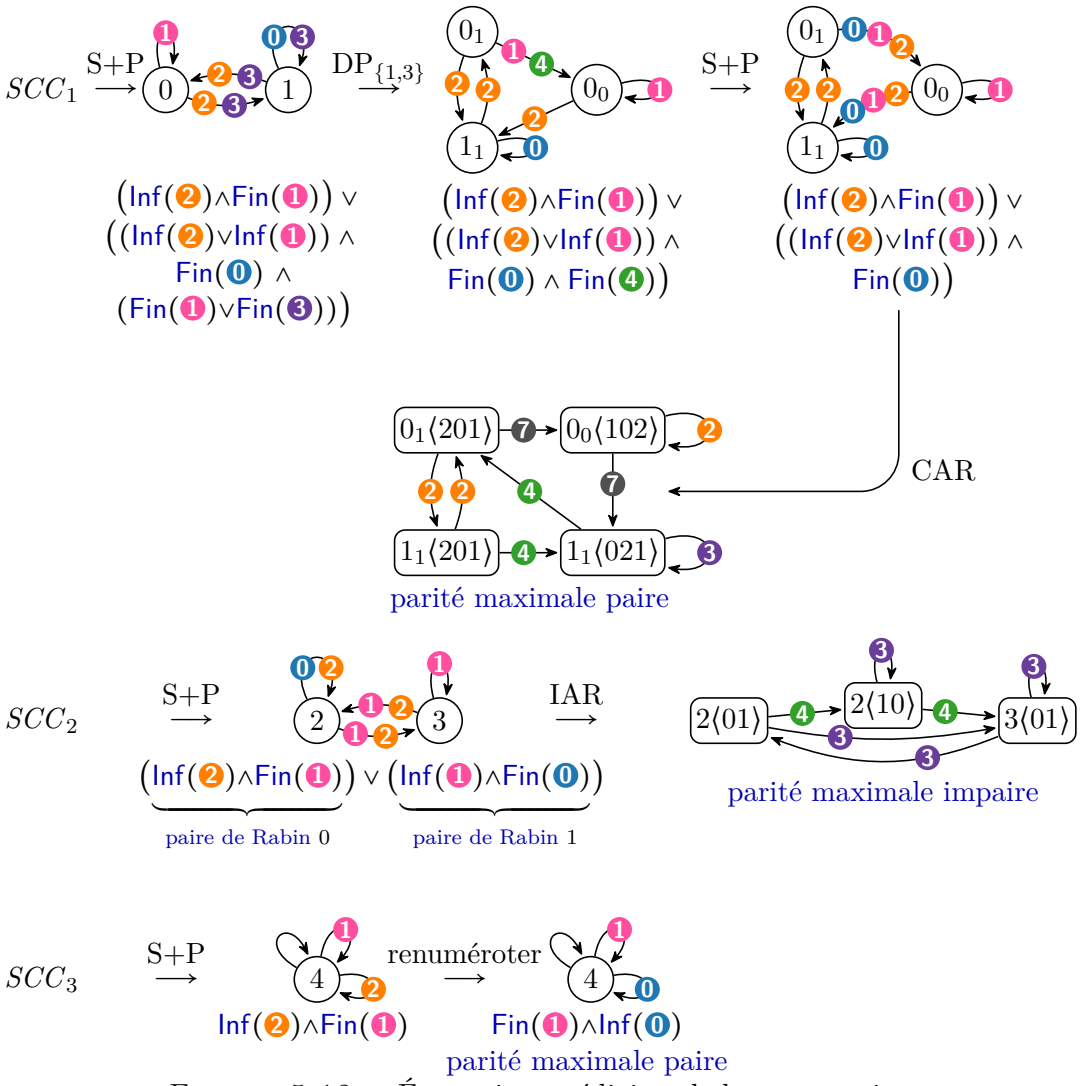


FIGURE 5.12. – Étapes intermédiaires de la construction

Remarque 41. *La différence avec la détection des automates parity-type est qu'il est facile de trouver les arêtes toujours acceptantes ou rejetantes dans un automate portant une condition de parité.*

On en conclut alors que si toutes les SCC de l'automate sont parity-type, alors il n'y aura pas besoin d'appliquer cette minimisation du nombre de couleurs à la fin de la construction. Il en est de même pour le cas où l'automate n'accepte aucun mot.

5.11.3. Simplification de condition

Nous allons maintenant décrire quelques simplifications de condition utilisées dans Spot. Ces transformations ont principalement pour but de réduire le nombre de couleurs de l'automate à traiter sans augmenter sa taille puisque la complexité de CAR est factorielle

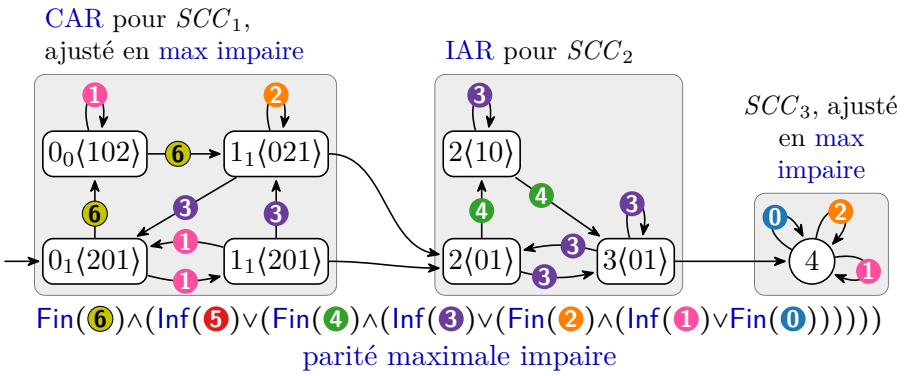


FIGURE 5.13. – Résultat de la paritisation de l'automate de la figure 5.11

en le nombre de couleurs.

Les règles que nous allons décrire s'appliquent sur un automate $\mathcal{A}$ portant une condition quelconque α et l'on note $\alpha[\beta \leftarrow \gamma]$ le remplacement dans α de toute occurrence de β par γ. Par exemple $\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \wedge \mathsf{Fin}(2)[\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \leftarrow \mathsf{Inf}(3)]$ est une règle de réécriture de la dégénéralisation partielle et le résultat est $\mathsf{Inf}(3) \wedge \mathsf{Fin}(2)$.

Nettoyage basique

La première simplification s'appuie sur une recherche de couleurs présentes sur toutes (respectivement aucune) les arêtes. Si une telle couleur c existe, alors $\mathsf{Inf}(c)$ sera forcément vrai (respectivement faux) et $\mathsf{Fin}(c)$ sera faux (respectivement vrai).

Pour une couleur c présente sur toutes les arêtes on a donc la possibilité de réécrire α comme $\alpha[\mathsf{Fin}(i) \leftarrow \bot][\mathsf{Inf}(i) \leftarrow \top]$ alors que pour une couleur c absence de l'automate on peut appliquer $\alpha[\mathsf{Fin}(i) \leftarrow \top][\mathsf{Inf}(i) \leftarrow \bot]$.

Fusion de couleurs inséparables

Une deuxième simplification s'appuie sur le fait que si deux couleurs sont telles que la présence de l'une sur une arête implique la présence de l'autre, alors une seule de ces couleurs est nécessaire.

Ainsi si toute arête portant c_1 porte également c_2 il est possible de réécrire α comme $\alpha[\mathsf{Fin}(c_1) \leftarrow \mathsf{Fin}(c_2)][\mathsf{Inf}(c_1) \leftarrow \mathsf{Inf}(c_2)]$.

On peut alors supprimer la couleur c_1 de l'automate.

Simplification des couleurs couvrantes

L'idée de base de la *simplification des couleurs couvrantes* est de trouver deux couleurs c_1 et c_2 telle que toute arête porte exactement une de ces couleurs.

Ainsi voir finiment c_1 et infiniment c_2 est équivalent à voir finiment souvent c_1. Le même type de raisonnement pour les différentes combinaisons de Fin et de Inf nous donne

la règle de réécriture suivante :

$$\alpha[\mathsf{Fin}(c_1) \wedge \mathsf{Inf}(c_2) \leftarrow \mathsf{Fin}(c_1)][\mathsf{Fin}(c_1) \wedge \mathsf{Fin}(c_2) \leftarrow \bot]$$
$$[\mathsf{Fin}(c_1) \vee \mathsf{Inf}(c_2) \leftarrow \mathsf{Inf}(c_2)][\mathsf{Inf}(c_1) \vee \mathsf{Inf}(c_2) \leftarrow \top]$$

Propagation unitaire (contribution)

La *propagation unitaire* est une simplification de la condition indépendante de l'automate s'appuyant sur le fait que $\mathsf{Inf}(i)$ et $\mathsf{Fin}(i)$ se comportent comme des littéraux positifs et négatifs (voir remarque 7). Ainsi s'ils apparaissent comme des clauses unitaires dans une conjonction ou une disjonction, alors ils peuvent être propagés aux autres clauses. Par exemple considérons la formule $\mathsf{Inf}(0) \vee (\mathsf{Fin}(0) \wedge \mathsf{Inf}(1))$. S'il y a besoin d'évaluer la deuxième clause, cela implique que $\mathsf{Inf}(0)$ est faux, ou de manière équivalente que $\mathsf{Fin}(0)$ est vrai. $\mathsf{Fin}(0)$ peut alors être remplacé par $\top$ et on obtient la condition $\mathsf{Inf}(0) \vee \mathsf{Inf}(1)$.

De manière générale, dans une formule de la forme $\mathsf{Inf}(i) \vee \beta$ ou $\mathsf{Fin}(i) \wedge \beta$, on peut remplacer β par $\beta[\mathsf{Inf}(i) \leftarrow \bot][\mathsf{Fin}(i) \leftarrow \top]$ alors que pour une formule de la forme $\mathsf{Inf}(i) \wedge \beta$ ou $\mathsf{Fin}(i) \vee \beta$ on peut remplacer β par $\beta[\mathsf{Inf}(i) \leftarrow \top][\mathsf{Fin}(i) \leftarrow \bot]$.

Fusion des conjonctions de Fin ou des disjonctions de Inf (contribution)

La *fusion* que nous allons décrire ici concerne les sous-formules de la forme $\beta = \mathsf{Inf}(i) \vee \mathsf{Inf}(j)$ (respectivement $\beta = \mathsf{Fin}(i) \wedge \mathsf{Fin}(j)$). L'idée est d'ajouter une nouvelle couleur k sur les arêtes portant i ou j et de remplacer la formule β par $\mathsf{Inf}(k)$ (respectivement $\mathsf{Fin}(k)$). Puisque notre but est de réduire le nombre de couleurs, cette opération ne peut être faite que si i et j ne sont pas présentes ailleurs dans la condition.

De manière plus générale, considérons une formule α où les couleurs $m_1, \ldots, m_n$ n'apparaissent qu'une fois dans α. Alors toute sous-formule de la forme $\mathsf{Inf}(j) \vee \bigwedge_{i=1}^{n} (\mathsf{Inf}(m_i) \vee \beta_i)$ peut être réécrite comme $\bigwedge_{i=1}^{n} (\mathsf{Inf}(m_i) \vee \beta_i)$ en remplaçant dans l'automate toutes les occurrences de j par $\{m_1, \ldots, m_n\}$. La même idée permet de supprimer $\mathsf{Fin}(j)$ d'une formule de la forme $\mathsf{Fin}(j) \vee \bigvee_{i=1}^{n} (\mathsf{Fin}(m_i) \wedge \beta_i)$

Acceptation basée sur les transitions

Une dernière simplification s'appuie sur la remarque 27 pour transformer un automate portant plus de couleurs que d'arêtes en un automate avec autant de couleurs que d'arêtes.

Remarque 42. *Pour un automate portant plus de couleurs que d'arêtes, appliquer TAR est équivalent à appliquer CAR après la dernière simplification.*

5.11.4. Conservation d'une SCC terminale

L'idée derrière la conservation d'une *SCC terminale* est de corriger le choix de la mémoire initiale pour les algorithmes de la famille LAR et les dégénéralisations.

Considérons l'exemple de la figure 5.14. La mémoire initiale est $\langle 0, 1, 2, 3 \rangle$. Si au lieu de choisir cette mémoire on avait décidé d'utiliser $\langle 3, 2, 1, 0 \rangle$, alors l'état $(0, \langle 0, 1, 2, 3 \rangle)$ n'aurait jamais été créé et nous aurions donc un automate à 3 états.

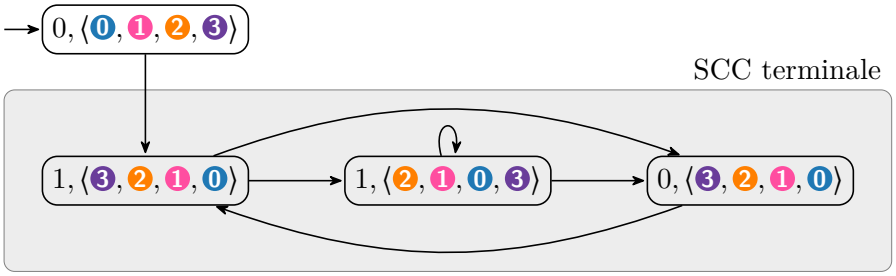


FIGURE 5.14. – Structure d'un automate de parité obtenu à partir de l'application de CAR sur un automate à deux états

De manière générale, cette optimisation est à mettre en lien avec de découpage en SCC de l'automate de départ. Puisque LAR est appliqué sur un sous-automate $\mathcal{A}'$ avec une seule SCC, alors chaque SCC terminale doit contenir au moins une copie de tous les états de $\mathcal{A}'$. On peut alors choisir de ne conserver que les états de n'importe quelle SCC terminale.

En pratique il s'agit donc de construire un automate avec LAR puis de déterminer quels états appartiennent à une des SCC terminales. Les SCC de l'automate résultant seront ensuite reconnectées à ces états.

Remarque 43. *Nous l'avons ici appliqué sur LAR mais la même idée peut être appliquée à la dégénéralisation partielle ou encore aux arbres de Zielonka.*

5.11.5. Recherche d'état existant

Dans la définition de CAR et IAR, nous avons indiqué qu'il est possible de choisir n'importe quel ordre de déplacement des éléments de la mémoire sans affecter la correction du résultat. KŘETÍNSKÝ et al. [43] ont suggéré de choisir un ordre de manière à obtenir lorsque c'est possible un état déjà créé. Ces états déjà créés seront désignés comme *compatibles*.

Un des éléments qu'ils n'ont pas décrit est le comportement à adopter dans le cas où plusieurs états compatibles existent.

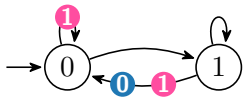
Deux types d'états compatibles seront utilisés. Un état compatible q sera le *plus ancien* si tous les autres états compatibles ont été créés après q lors de la construction de l'automate de parité. À l'inverse un état compatible q sera le *plus récent* si tout autre état compatible a été créé avant q.

Pour comprendre l'idée derrière la recherche du plus récent état compatible, considérons le cas de la figure 5.15. Dans cet exemple, lorsque l'on est dans une copie de l'état 1 on peut aller vers n'importe quelle copie de l'état 0 en déplaçant 0 et 1 dans un ordre adapté.

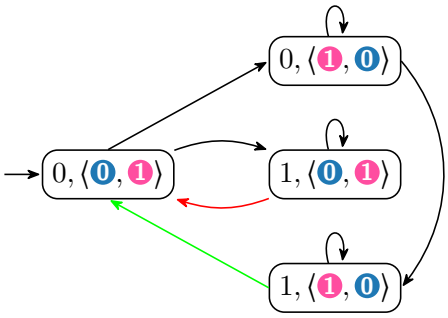
Dans les figures 5.15b et 5.15c, les états sont créés dans l'ordre suivant :

— $(0, \langle 0, 1 \rangle)$;

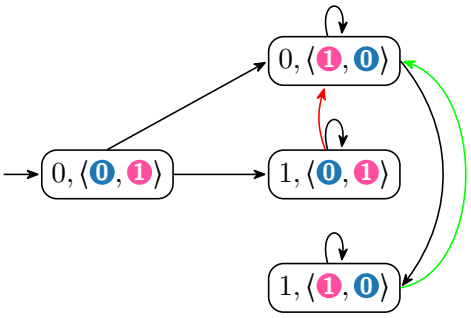
— $(0, \langle 1, 0 \rangle)$;



(a) Automate à paritiser



(b) Automate obtenu en utilisant le plus ancien état compatible.



(c) Automate obtenu en utilisant le plus récent état compatible.

FIGURE 5.15. – Construction CAR avec la recherche du plus récent et du plus ancien état compatible. Les arêtes colorées en rouge et en vert sont celles pour lesquelles il est possible de faire un choix.

— $(1, \langle 0, 1 \rangle)$;

— $(1, \langle 1, 0 \rangle)$.

Deux cas sont décrits. Le premier consiste à rechercher le plus ancien état compatible alors que le second recherche le plus récent.

Une différence se situe dans le choix de la destination de l'arête partant de $(1, \langle 1, 0 \rangle)$ et devant aller vers une copie de l'état 0. Dans le premier cas puisque $(0, \langle 0, 1 \rangle)$ a été le premier état créé, il sera l'état compatible choisi. Cela implique que les quatre états de l'automate sont alors dans la SCC terminale.

À l'inverse choisir l'état compatible le plus récent permet de ne conserver que deux états dans la SCC terminale.

On peut remarquer qu'utiliser l'état compatible le plus récent dépend de l'ordre de construction des états. Supposons que $(0, \langle 1, 0 \rangle)$ soit créé après $(1, \langle 0, 1 \rangle)$. Dans ce cas l'arête rouge n'aurait pas pu avoir le premier comme destination (puisque pas encore créé). L'idée peut alors être d'attendre la fin de la construction pour changer toutes les arêtes de manière à ce qu'elles aillent vers l'état compatible le plus récent.

5.11.6. Heuristique sur l'ordre des couleurs

Alors que la recherche d'état existant s'appuie sur des états déjà créés, nous pouvons chercher à déterminer un moyen d'ordonner les couleurs avant la construction.

Nous allons ici voir une manière de créer un tel ordre en analysant les arêtes entrantes d'un état.

(a) (b)

FIGURE 5.16. – Automates à partir desquels on peut extraire un ordre de déplacement des couleurs

Considérons l'automate de la figure 5.16a. La question est ici de savoir si lors du passage de l'état 0 à l'état 1 il faut mettre 0 puis 1 au début de la mémoire ou l'inverse.

Les arêtes qui vont vers l'état 1 portent les ensembles de couleurs $\{0,1\}$ et $\{1\}$.

Le deuxième ensemble de couleur implique qu'il devra exister une mémoire commençant par 1 ($\langle 1,0\rangle$ ici). Ainsi si l'on décide de déplacer 1 puis 0 (ce qui donne la mémoire $\langle 0,1\rangle$) on se retrouvera avec deux états différents alors que choisir l'autre ordre aurait donné une unique mémoire commune.

De manière générale, il s'agit pour chaque état q de partitionner l'ensemble des colorations des arêtes qui y vont. Chaque classe C_i de cette partition doit être telle que pour toute paire c_1, c_2 d'éléments (distincts) de C_i, $c_1 \subseteq c_2$ ou $c_2 \subseteq c_1$.

Chaque classe est alors totalement ordonnée par inclusion. Il reste alors à construire à partir de cette partition un ordre.

Considérons la classe $C = \{\{0\},\{0,1\},\{0,1,2,3\}\}$ associée à un état q. Puisque tous les éléments ont 0 en commun, il s'agira de la couleur la plus à gauche. Si on supprime 0 des éléments de notre classe, alors les ensembles non-vides partagent 1 et il s'agira donc du deuxième élément dans la mémoire. En supprimant 1 des ensembles il ne nous reste que 2 et 3 à placer. Ces deux couleurs peuvent être placées dans n'importe quel ordre.

On choisira donc de déplacer 3 puis 2 puis 1 puis 0 lorsque l'on traite une arête allant vers q et qui porte un des ensembles de couleurs de C.

Exemple 16. *Considérons l'automate de la figure 5.16b. On peut en tirer les informations suivantes :*

- *Pour l'état 0, la seule arête qui y va porte la couleur 0 donc toute mémoire qui sera associée à l'état 0 pourra porter 0 à gauche ;*
- *Pour l'état 1, les arêtes qui y vont portent les couleurs $\{0,1\}$ et $\{1,2\}$. Ces deux ensembles sont incomparables (au sens de l'inclusion) donc on ne peut pas fixer d'ordre avec cette méthode ;*
- *Pour l'état 2, les arêtes qui y vont portent les couleurs $\{0,2\}$ et $\{2\}$. Puisque $\{2\} \subset \{0,2\}$, on peut choisir de déplacer 0 puis 2. Ainsi toute mémoire associée à l'état 2 portera 2 à sa gauche.*

(a) Automate dont les couleurs ne sont pas propagées

(b) Résultat de la propagation de l'automate de la figure 5.17a

FIGURE 5.17. – Application de la propagation des couleurs basée sur la structure de l'automate

Remarque 44. *Dans l'exemple 16, plus d'informations peuvent être extraites. En effet nous venons d'indiquer qu'une mémoire associée à 0 peut toujours commencer par* 0*. Cependant, on sait que toute mémoire associée l'état 2 commencera par* 2 *(si on l'impose). On peut donc n'associer l'état 2 qu'à* ⟨2, 0, 1⟩ *et* ⟨2, 1, 0⟩*.*

Puisque l'ensemble des mémoires associées à l'état 0 correspond à l'ensemble des mémoires associées à l'état 2 où l'on déplace 0 *à gauche, cela signifie que* ⟨0, 2, 1⟩ *peut être la seule mémoire associée à l'état 0.*

Cette analyse n'est pas implémentée dans Spot.

5.11.7. Propagation des couleurs

Nous allons ici parler de la *propagation des couleurs*. Il s'agit d'une optimisation visant à donner plus de choix à la recherche d'état existant ou encore à aider les diverses simplifications de l'automate (nettoyage basique, fusion de couleurs inséparables, simplification des couleurs couvrantes). Même si elle est aussi applicable dans le cadre de la dégénéralisation (partielle ou non) pour augmenter la probabilité de pouvoir passer plusieurs niveaux à la fois, nous allons ici la présenter à travers l'application de CAR.

La propagation telle qu'elle est implémentée dans Spot regroupe deux transformations. La première se base sur la structure de l'automate pour ajouter des couleurs aux arêtes alors que la seconde traite indépendamment chaque arête et lui ajoute des couleurs en s'appuyant sur la condition d'acceptation. Dans les deux cas les couleurs sont ajoutées de manière à ne pas changer le langage associé à l'automate.

Remarque 45. *Avec la fonction* `to_parity` *de Spot, l'option* `propagate_col` *indique s'il faut une propagation pour la simplification et LAR. Une autre propagation est effectuée lors de la dégénéralisation partielle mais n'est pas désactivable.*

Propagation basée sur la structure de l'automate

Nous allons illustrer la propagation sur l'automate de la figure 5.17a. La première étape est de considérer l'arête allant de l'état 1 à l'état 0 qui n'est pas colorée. Pour voir cette dernière infiniment souvent, il faut passer infiniment souvent par l'arête allant de

Algorithme 5 : Algorithme de propagation des couleurs basée sur la structure de l'automate.

Entrée : Un automate $\mathcal{A}$
Sortie : L'automate $\mathcal{A}$ dont les couleurs sont propagées

faire
  pour $q \in Q$ **faire**
    $In = \{(s,a,c,d) \in \delta \mid d = q \text{ et } \mathrm{SCC}(s) = \mathrm{SCC}(d) \text{ et } s \neq d\}$
    $Out = \{(s,a,c,d) \in \delta \mid s = q \text{ et } \mathrm{SCC}(s) = \mathrm{SCC}(d) \text{ et } s \neq d\}$
    $CommonColsIn = \bigcap_{(s,a,c,d)\in In} c$
    $CommonColsOut = \bigcap_{(s,a,c,d)\in Out} c$
    pour $(s,a,c,d) \in Out$ **faire**
      Remplacer c par $c \cup CommonColsIn$
    pour $(s,a,c,d) \in In$ **faire**
      Remplacer c par $c \cup CommonColsOut$
tant que *la dernière itération a modifié l'automate*
retourner $\mathcal{A}$

0 à 1. Puisque cette dernière porte 0, l'arête de 1 à 0 est visitée infiniment souvent si 0 est visité infiniment souvent. Ainsi on peut lui associer 0. En appliquant la même idée on peut ajouter à l'arête allant de l'état 1 à l'état 2 la couleur 1 et 0 à celle allant de l'état 2 à l'état 1. On obtient alors l'automate de la figure 5.17b. Cette procédure est décrite dans l'algorithme 5.

Propagation basée sur la condition

Le deuxième type de propagation étudie la condition pour savoir si pour un ensemble c de couleurs, il est possible de trouver un sur-ensemble c' de c tel que le remplacement de c par c' ne change pas le langage associé à l'automate.

Les conditions que nous allons traiter ici sont celles de la forme $(\bigwedge_{i\in\{1,\dots,n\}} \mathsf{Inf}(y_i)) \vee f(x_1,\dots,x_m)$ ou $(\bigvee_{i\in\{1,\dots,n\}} \mathsf{Fin}(y_i)) \vee f(x_1,\dots,x_m)$.

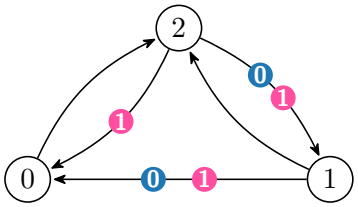
Alors à une arête portant les couleurs $\{y_1,\dots,y_n\}$ on peut ajouter n'importe quel sous-ensemble des couleurs de $\{x_1,\dots,x_m\} \setminus \{y_1,\dots,y_n\}$.

Exemple 17 (Propagation basée sur la condition associée à la propagation basée sur la structure)**.** *Considérons l'automate de la figure 5.18a.*

L'arête de l'état 2 à l'état 1 porte les couleurs 0 *et* 1*. On peut donc appliquer la propagation basée sur la condition et ajouter* 2*. En effet, visiter cette arête infiniment souvent implique que la partie* $\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)$ *est satisfaite et* 2 *peut alors être ajouté.*

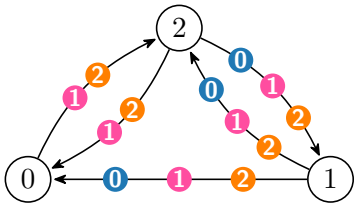
Appliquer une étape de propagation basée sur la structure permet ensuite de propager les couleurs tel que nous l'avons vu avec l'exemple 5.17. Nous obtenons alors l'automate de la figure 5.18b.

Remarque 46. *Puisque la couleur* 2 *est présente sur toutes les arêtes, la partie* $\mathsf{Fin}(2)$ *de la condition n'est pas possible,* $\mathsf{Fin}(2)$ *peut être supprimée de la condition et l'automate*



$(\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)) \vee \mathsf{Fin}(2)$

(a) Automate à traiter



$(\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)) \vee \mathsf{Fin}(2)$

(b) Automate dont les couleurs ont été propagées

FIGURE 5.18. – Combinaison des deux méthodes de propagation

porte alors la condition $\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)$. *La propagation a donc permis d'appliquer une règle de simplification supplémentaire.*

Impact sur la recherche d'état existant avec CAR

Nous allons maintenant illustrer l'impact de la propagation sur la recherche d'état existant en s'appuyant sur l'exemple de la figure 5.18.

Commençons par traiter l'automate de la figure 5.18a. Supposons que le premier état créé est $(0, \langle 2, 1, 0\rangle)$. La première étape est de créer une copie de l'état 2 qui portera la même mémoire. Traiter l'arête allant de l'état 2 vers l'état 0 implique de créer un état $(0, \langle 1, 2, 0\rangle)$. Il n'y a donc eu aucun choix à faire et 3 états ont été créés.

Appliquons la même construction sur l'automate propagé. Comme avant il s'agit de créer une copie de l'état 2. L'arête porte les couleurs 1 et 2. Choisissons alors de garder la mémoire $\langle 2, 1, 0\rangle$.

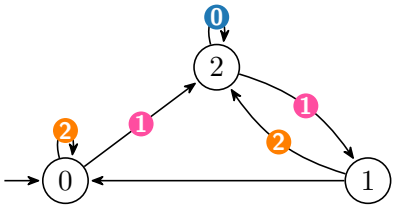
En traitant l'arête allant de l'état 2 à l'état 0, on voit maintenant qu'il existe un choix d'ordre de déplacement permettant d'obtenir l'historique associé à l'état initial. Il n'y a alors pas besoin de créer de nouvel état.

5.11.8. Préfixe de parité

Construction basée sur l'analyse de la condition

Pour comprendre l'idée derrière le *préfixe de parité*, appuyons-nous sur une condition de la forme $\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge \mathsf{Fin}(2))$. Sans même étudier l'automate, on sait que si un cycle passe par une arête portant 0, alors il sera acceptant. On sait également que dans l'automate de parité final, chaque copie de cette arête devra respecter les mêmes propriétés. L'idée est alors de supprimer ces arêtes, de paritiser les sous-automates restants avant de rajouter des copies de ces arêtes supprimées.

Remarque 47. *Il s'agit d'une construction récursive. Après avoir retiré les arêtes toujours acceptantes, on peut appliquer la même idée sur les arêtes toujours rejetantes comme nous l'avions fait pour les automates parity-type.*



$\mathsf{Fin}(0) \wedge (\mathsf{Inf}(1) \wedge \mathsf{Inf}(2))$

FIGURE 5.19. – Automate pour lesquel il est possible d'extraire un préfixe de parité

Exemple 18. *Considérons l'automate de la figure 5.19. La condition est de la forme* $\mathsf{Fin}(0) \wedge \beta$. *On va donc :*

- *Supprimer les arêtes portant* 0 *de l'automate ;*
- *Paritiser l'automate obtenu en un automate* $\mathcal{P}$ *portant une condition* α' *;*
- *Ajouter une boucle à chaque copie de l'état* 2 *coloriée par une couleur* c *rejetante supérieure à toutes les couleurs de* α' *;*
- *Assigner à* $\mathcal{P}$ *la condition* $\mathsf{Fin}(c) \wedge \alpha'$.

Construction basée sur l'analyse de la structure de l'automate

Alors que la construction précédente se base sur une analyse de la condition, nous allons ici généraliser l'algorithme utilisé pour détecter si un automate est parity-type.

Nous avons vu dans la section 5.10 qu'il est possible d'extraire une suite d'ensembles d'arêtes toujours acceptantes ou rejetantes. Dans le cas des automates parity-type, nous obtenions un résultat si toutes les arêtes pouvaient être recoloriées.

Ici, nous retirons simplement toutes les arêtes qui ont été coloriées par cette procédure et appliquons des paritisations sur les parties restantes de l'automate.

5.12. Transformation basée sur un arbre de Zielonka

Cette dernière section décrit la création de DPA à l'aide d'arbres de Zielonka. Il s'agit d'un algorithme proche de ceux de la famille LAR puisqu'il implique la création d'un automate reconnaissant la condition de l'automate de départ et d'en faire le produit avec ce même automate. Cependant il y a ici une garantie d'optimalité sur la taille de l'automate reconnaissant la condition.

La présentation de cet algorithme se fera en deux parties. La première consistera à décrire la création d'un arbre de Zielonka à partir d'une condition quelconque. La seconde décrira comment cet arbre est utilisé pour créer un automate de parité.

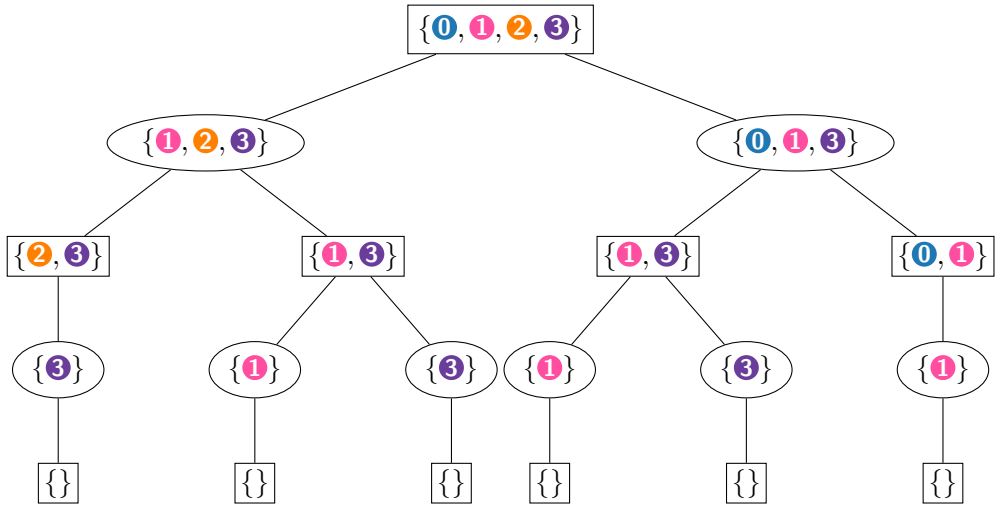


FIGURE 5.20. – Arbre de Zielonka décrit dans l'exemple 19 pour la condition $\mathsf{Fin}(0) \wedge \mathsf{Inf}(1) \wedge (\mathsf{Inf}(2) \vee \mathsf{Fin}(3)) \vee (\mathsf{Inf}(0) \vee \mathsf{Fin}(1)) \wedge \mathsf{Fin}(2) \wedge \mathsf{Inf}(3)$

5.12.1. Arbre de Zielonka

Commençons par décrire les arbres de Zielonka. Il s'agit d'un arbre dans lequel chaque nœud porte un ensemble de couleurs. De plus à chaque *nœud* est associé un Booléen indiquant si l'ensemble des couleurs portées par ce nœud est acceptant ou non.

Définition 67 (Arbre de Zielonka)**.** *Soit α une condition d'acceptation portant sur un ensemble M de couleurs. Un* arbre de Zielonka *associé à α est un arbre (V, E) tel que chaque nœud porte un ensemble de couleurs et un Booléen construit ainsi :*

- *La racine est (M, b) où b est vrai si et seulement si $M \vDash \alpha$;*
- *Si $n = (M', \top)$ est un nœud déjà construit, alors pour tous les sous-ensembles maximaux $M_1, \ldots, M_k$ de M' rejetants, on ajoute un fils $(M_i, \bot)$ à n ;*
- *Si $n = (M', \bot)$ est un nœud déjà construit, alors pour tous les sous-ensembles maximaux $M_1, \ldots, M_k$ de M' acceptants, on ajoute un fils $(M_i, \top)$ à n.*

Remarque 48. *Dans un arbre de Zielonka, deux nœuds de branches différentes peuvent porter la même étiquette. Même s'il est possible d'appliquer une mémoïsation lors de la construction de l'arbre, ces nœuds ne peuvent pas être fusionnés.*

Pour des questions de lisibilité et de manière à avoir une représentation proche de celle de CASARES et al. [14], un nœud portant $\top$ sera ovale alors qu'un nœud portant $\bot$ sera rectangulaire.

Exemple 19 (Arbre de Zielonka)**.** *Construisons*[1] *l'arbre de Zielonka de la condition* $\mathsf{Fin}(0) \wedge \mathsf{Inf}(1) \wedge (\mathsf{Inf}(2) \vee \mathsf{Fin}(3)) \vee (\mathsf{Inf}(0) \vee \mathsf{Fin}(1)) \wedge \mathsf{Fin}(2) \wedge \mathsf{Inf}(3)$ *représenté dans la figure 5.20.*

1. Une version interactive de cet arbre est disponible à l'adresse `https://spot-sandbox.lrde.epita.fr/notebooks/examples%20(read%20only)/zlktree.ipynb`

Puisqu'une exécution voyant infiniment souvent les couleurs $\{0, 1, 2, 3\}$ *est rejetante, alors la racine portant ce même ensemble est rejetant.*

Il s'agit ensuite de trouver les sous-ensembles maximaux de $\{0, 1, 2, 3\}$ *acceptants. Ces deux ensembles sont* $\{1, 2, 3\}$ *et* $\{0, 1, 3\}$ *et deux nouveaux nœuds portant ces ensembles forment la liste des fils de la racine.*

Réitérer cette procédure conduit à l'obtention de l'arbre de la figure 5.20.

5.12.2. Construction d'un automate de parité à partir d'un arbre de Zielonka

Notre objectif est maintenant de construire un automate de parité à l'aide d'un arbre de Zielonka. Avant de définir plus formellement cette construction, nous allons décrire sur un exemple cette transformation.

Un point à mettre en avant est que contrairement aux constructions présentées précédemment, nous allons obtenir un automate à parité minimale dans le but de rester proche de la définition donnée par Casares et al. [14]. Comme nous l'avons vu, si un automate à condition de parité maximale est requis, il peut être obtenu facilement à partir d'un automate de parité minimale (voir remarque 15).

Décrivons de manière formelle cette construction.

Pour cela plaçons-nous dans le cadre de la paritisation d'un TELA $\mathcal{A} = (Q, \Sigma, q_0, \delta, \alpha)$ où la condition α est associée à l'arbre de Zielonka $\mathcal{Z}_\mathcal{A}$.

Notation 8. *Étant donné un nœud* $x = (C, b)$ *d'un arbre de Zielonka, on note* $\Gamma(x)$ *la fonction associant à* x *l'ensemble* C*.*

Étant donnée une arête $e = (q \xrightarrow{\ell,c} q')$ et une feuille x de $\mathcal{Z}_\mathcal{A}$, on définit $\mathit{Support}(x, e)$ comme étant le plus proche ancêtre y de x dans $\mathcal{Z}_\mathcal{A}$ tel que $c \subseteq \Gamma(y)$.

Si $\mathit{Support}(x, e) \neq x$ et n'est pas une feuille de $\mathcal{Z}_\mathcal{A}$, désignons par z l'unique enfant de $\mathit{Support}(x, e)$ qui est un ancêtre de x et $y_1, \ldots, y_s$ une énumération de gauche à droite des nœuds de $\mathrm{Children}_{\mathcal{Z}_\mathcal{A}}(\mathit{Support}(x, e))$.

On définit $\mathit{NextBranch}(x, e)$ comme :

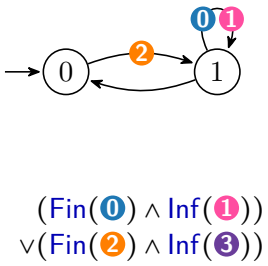
$$\begin{cases} \mathit{Support}(x, e) & \text{si } \mathit{Support}(x, e) = x \text{ ou } \mathit{Support}(x, e) \text{ est une feuille de } \mathcal{Z}_\mathcal{A} \text{ ;} \\ y_1 & \text{si } z = y_s \text{ ;} \\ y_{j+1} & \text{si } z = y_j, 1 \leq j < s. \end{cases}$$

Étant donné un automate $\mathcal{A} = (Q, \Sigma, q_0, \delta, \alpha)$, on définit l'*automate de Zielonka* $\mathcal{Z}(\mathcal{A}) = (Q', \Sigma, q_0', \delta', \alpha')$ ainsi :

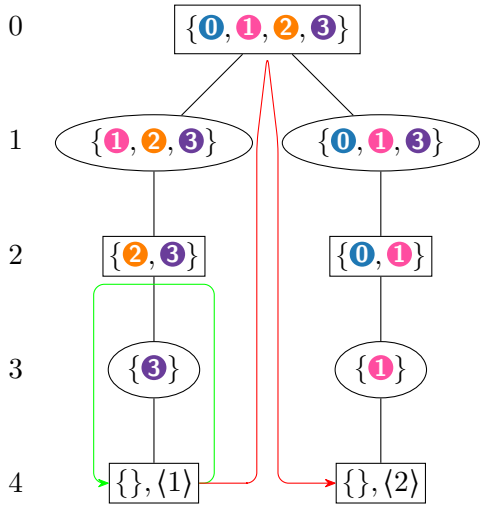
États Les états de $\mathcal{Z}(\mathcal{A})$ sont de la forme (q, x) où $q \in Q$ et x est une feuille (un identifiant dans la description précédente) dans l'arbre associé à α :

$$Q' = \bigcup_{q \in Q} Q \times \mathrm{Leaves}_{\mathcal{Z}_\mathcal{A}}$$

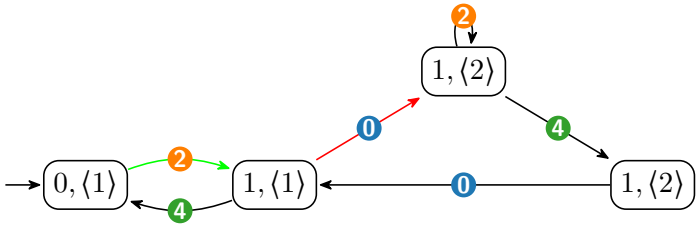
État initial L'état initial q_0' de $\mathcal{Z}(\mathcal{A})$ est de la forme (q_0, x_0) où x_0 est la feuille la plus à gauche de $\mathcal{Z}_\mathcal{A}$.



(a) Automate à paritiser



(b) Arbre de Zielonka associé à la condition de l'automate de la figure 5.21a. Les flèches colorées représentent certains parcours de l'arbre pour créer le DPA de la figure 5.21c.



(c) Construction d'un automate de parité minimale impaire à partir de l'automate de la figure 5.21a et de l'arbre de la figure 5.21b

FIGURE 5.21. – Exemple de création d'un DPA à l'aide d'un arbre de Zielonka

Transitions Pour chaque transition $(q \xrightarrow{\ell_i,c} q')$ de δ et tout état $(q,x) \in Q'$, on a une transition $(q,x) \xrightarrow{\ell_i,c'} (q',y)$ dans δ' telle que y est le descendant le plus à gauche de *NextBranch*(x,e) dans $\mathcal{Z}_\mathcal{A}$ alors que c est *Depth*(*Support*(x,e)) si l'ensemble de couleurs associé à la racine est acceptant et *Depth*(*Support*(x,e)) + 1 sinon.

Condition de parité La condition α' est une condition de parité minimale paire.

Théorème 9 (Optimalité de la construction sur la taille de l'automate reconnaissant la condition, [14])**.** *Étant donné un automate portant une condition de Muller, l'utilisation de l'arbre de Zielonka permet d'obtenir l'automate minimal reconnaissant cette condition, tant au niveau du nombre d'états que de couleurs.*

Appliquons maintenant cette construction sur l'automate de la figure 5.21a. Ce dernier porte une condition de Rabin avec deux paires dont l'arbre de Zielonka est donné dans

la figure 5.21b.

Afin de pouvoir manipuler cet arbre, nous avons ajouté des identifiants aux feuilles ainsi que des niveaux.

L'idée de la construction est de créer un automate où les états associent un état de l'automate de départ et un identifiant de feuille de l'arbre. La première étape de la construction du DPA de la figure 5.21c est la création d'un état initial $(0, \langle 1 \rangle)$.

Traitons alors l'arête allant de l'état 0 à l'état 1 dans l'automate de la figure 5.21a. Celle-ci porte la couleur 2. Plaçons-nous dans la feuille $\langle 1 \rangle$ de l'arbre de la figure 5.21b et regardons jusqu'à quel nœud il faut remontrer pour voir la couleur 2. Celui-ci est au niveau 2. À partir de ce nœud on va alors redescendre vers la feuille de la branche suivante. Puisqu'ici il n'y a qu'une seule feuille accessible depuis ce nœud, on arrive dans la feuille $\langle 1 \rangle$. Cela correspond au chemin vert. On doit alors créer un état étiqueté par $(1, \langle 1 \rangle)$. Puisque nous sommes remontés jusqu'à un nœud du niveau 2, la couleur 2 est associée à l'arête que nous ajoutons entre ces deux états.

À partir de cet état $(1, \langle 1 \rangle)$, traitons la boucle de l'état 1. En partant dans l'arbre de la feuille $\langle 1 \rangle$, on va remonter jusqu'à voir les couleurs 0 et 1. Cela signifie qu'il faut remonter jusqu'à la racine. À partir de cette racine, on va aller dans la branche suivante (celle de droite) et redescendre jusqu'à la feuille $\langle 2 \rangle$. Cela correspond au chemin rouge. Un état $(1, \langle 2 \rangle)$ est alors créé et l'arête porte la couleur 0.

Enfin décrivons le cas de l'arête étiquetée par 4 dans l'automate résultant. L'état de départ est $(0, \langle 2 \rangle)$. Depuis cette feuille, il est impossible de remonter à un niveau supérieur. La feuille atteinte sera alors forcément celle de départ. Puisque l'on n'est pas remonté, la couleur 4 sera émise.

Puisque la racine porte un ensemble rejetant, la condition est une condition de parité minimale impaire.

5.12.3. Classes particulières d'arbres de Zielonka

Le but de cette partie est de décrire le lien entre certaines classes de conditions et la structure des arbres de Zielonka associés. Cela nous permettra également de voir comment une condition équivalente à une condition de parité peut être transformée de manière à obtenir une condition de parité.

Condition de parité

Commençons par l'étude de la condition de parité minimale paire $\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(2))$.

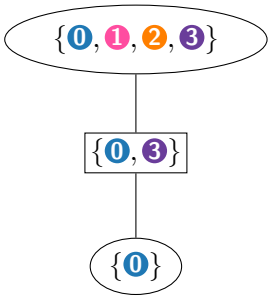
La construction de cet arbre présenté dans la figure 5.22b se fait en quatre étapes :

— La racine portant toutes les couleurs est acceptante ;

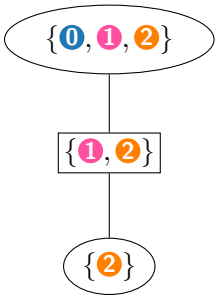
— Son fils doit être rejetant donc doit supprimer 0 ;

— À partir du nœud $\{1, 2\}$ il faut supprimer 1 pour obtenir un ensemble acceptant ;

— Le sous-ensemble vide est rejetant.



(a) Arbre de Zielonka associé à $\mathsf{Inf}(2) \vee \mathsf{Inf}(1) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(0))$



(b) Arbre de Zielonka associé à $\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(2))$

FIGURE 5.22. – Arbre de Zielonka associé à différentes conditions équivalentes à des conditions de parité

On a vu qu'à chaque étape il fallait supprimer la plus petite couleur. Ainsi lors du passage d'un nœud acceptant à un nœud rejetant on a vu une couleur qui devait être vue infiniment souvent alors que pour l'inverse c'est une couleur qui doit apparaître finiment souvent.

De manière générale, une condition est équivalente à une condition de parité si l'arbre de Zielonka qui lui est associé aura cette structure (l'existence d'au plus un fils). De plus, cette structure permet de réécrire de telles conditions en condition de parité.

Appliquons maintenant cette idée sur la condition $\mathsf{Inf}(2) \vee \mathsf{Inf}(1) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(0))$. L'arbre de Zielonka associé est donné dans la figure 5.22a.

La racine y est acceptante et son fils est obtenu en supprimant 1 et 2 de ses couleurs. Ce sont donc des couleurs qui devront être vues infiniment souvent.

À partir du nœud portant $\{0, 3\}$ on rejoint son fils en supprimant la couleur 3 qui doit donc être rejetante.

Enfin on conclut de la même manière que 0 doit être acceptant.

La condition peut alors être vue comme $\mathsf{Inf}(1 \vee 2) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(0))$.

Remplacer 1 et 2 par 0, 3 par 1 et 0 par 2 dans la condition et l'automate permet l'obtention d'un automate de parité minimale paire.

Remarque 49. *Sur cet exemple une simplification de la condition était suffisante puisqu'il s'agissait principalement de réécrire la disjonction de* Inf *avant une réattribution des couleurs.*

Cependant il est possible d'appliquer ce raisonnement sur des conditions plus complexes comme $\mathsf{Inf}(1) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(0) \vee (\mathsf{Fin}(0) \wedge \mathsf{Inf}(2) \wedge \mathsf{Inf}(1)))$.

Remarque 50. *Dans la procédure de paritisation générale, nous avons indiqué que l'on cherche à détecter si la condition de départ est équivalente à une condition de parité. En pratique, nous utilisons un arbre de Zielonka. Prenons comme exemple l'arbre de la figure 5.22a. L'idée est que si dans l'automate de départ, on remplace les* 1 *et* 2 *par* 2*,* 3 *par* 1*, alors on obtient un automate de parité.*

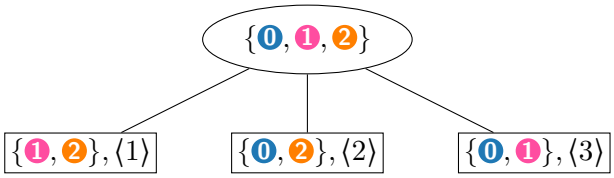


FIGURE 5.23. – Arbre de Zielonka associé à Inf(0) ∧ Inf(1) ∧ Inf(2)

Condition de Büchi généralisée

Le but ici est de montrer l'équivalence entre l'utilisation des arbres de Zielonka et la dégénéralisation pour obtenir un automate portant une condition de Büchi à partir d'un automate de Büchi généralisé.

Avec une condition de Büchi généralisée portant sur k couleurs, un arbre de Zielonka est composé d'une racine acceptante et d'un ensemble de k fils associés aux k moyens de retirer une couleur de la condition.

Considérons le cas de Inf(0) ∧ Inf(1) ∧ Inf(2).

L'arbre de Zielonka associé est donné dans la figure 5.23.

Sans en faire la construction, décrivons comment va être obtenu un automate de Büchi à l'aide de cet arbre. À partir de la feuille ⟨1⟩, il faut voir la couleur 0 pour passer au nœud ⟨2⟩ et donc émettre 0 dans le résultat.

À partir du nœud ⟨2⟩ il faut voir 1 pour passer au nœud ⟨3⟩ et pour passer du nœud ⟨3⟩ au nœud ⟨1⟩, il faut voir 2. Chaque nœud peut alors être vu comme un niveau dans l'automate de la dégénéralisation.

Remarque 51. *Alors que par définition la dégénéralisation va émettre 0 en allant du dernier au premier niveau, cette procédure va émettre 0 à chaque passage de "niveau", ce qui reste équivalent.*

Condition de Rabin

Le dernier type d'arbre qui est décrit est associé aux conditions de Rabin (et peut donc être adapté aux conditions de Streett).

Pour une condition équivalente à une condition de Rabin, l'arbre de Zielonka associé est tel que tout nœud acceptant a au plus un fils.

Notre exemple s'appuiera sur la condition (Fin(0) ∧ Inf(1)) ∨ (Fin(2) ∧ Inf(3)).

L'arbre d'une telle condition est donné dans la figure 5.24 et s'appuie sur le raisonnement suivant :

— La racine est rejetante ;

— Depuis un nœud acceptant, le seul moyen d'obtenir un ensemble de couleur rejetant est de supprimer une couleur dans un Inf qui n'est associé à aucun Fin (qui existe par construction) ;

— Depuis un nœud rejetant on supprime une couleur présente dans un Fin (qui existe par construction).

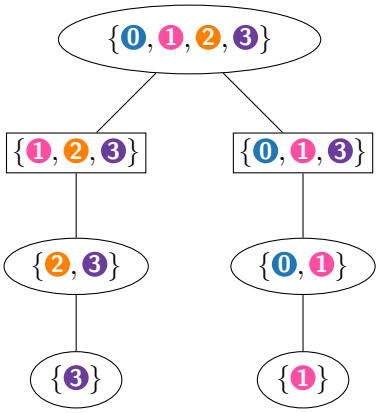


FIGURE 5.24. – Arbre de Zielonka associé à $(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(3))$

5.13. Évaluation expérimentale

Cette section vise à mettre en évidence l'impact des options de la procédure de paritisation basée sur LAR (`to_parity`).

Les mesures ont été effectuées à l'aide de `benchexec` [7] 3.10 avec une limite de 120 secondes et 6000 Mo de RAM et Spot a été configuré pour supporter au plus 32 couleurs.

Parmi les 1462 automates utilisés 727 proviennent de la traduction par Spot [2] d'une partie des 946 formules LTL issues de la SYNTCOMP [39] et 735 par Owl [3]. Supprimer les automates de parité conduit à l'obtention de 359 automates produits par `ltl2tgba` et 543 produits par Owl.

Les automates portant une condition de parité et `to_parity` vont simplement être copiés et ils ne seront donc pas considérés dans les résultats.

Dans toute cette section, une exécution A sera considérée comme plus rapide qu'une exécution B si la durée de B est supérieure à la durée de A multipliée par 1,1. Ceci a pour but de limiter l'impact du bruit de mesure.

Deux types de problèmes de mémoire peuvent arriver. Le premier est le dépassement de la quantité de mémoire autorisée. Le second correspond au cas où le système ne peut pas allouer de mémoire au programme. Ce second cas n'est pas géré par `benchexec` et est désigné comme une erreur. Sauf mention contraire, le terme erreur désignera donc ce cas.

Les optimisations seront désignées par leur nom dans Spot et ce lien est donné dans la table 5.1.

2. `ltl2tgba -extra-options=simul=0,ba-simul=0,det-simul=0,tls-impl=1,wdba-minimize=2 -G -D` avec une limite de 600 secondes
3. `owl ltl2dela` avec une limite de 600 secondes

Nom	Option	Référence
CAR	`--car`	sec. 5.3
IAR	`--iar`	sec. 5.6
TAR	`--tar`	sec. 5.4
Imposition de l'ordre	`--force_order`	sec. 5.11.6
SCC terminale	`--bscc`	sec. 5.11.4
Recherche d'état existant	`--search_ex`	sec. 5.11.5
Recherche du plus récent état compatible	`--use_last`	sec. 5.11.5
Recherche d'état compatible a posteriori	`--use_last_post_process`	sec. 5.13.2
Propagation des couleurs	`--propagate_col`	sec. 5.11.7
Simplification de l'automate	`--acc_clean`	sec. 5.11.3
Test d'équivalence à une condition de parité	`--parity_equiv`	
Suppression d'un préfixe de parité	`--parity_prefix`	sec. 5.11.8
Dégénéralisation partielle	`--partial_degen`	sec. 5.8
Gestion des automates de Rabin qui sont Büchi-type	`--rabin_to_buchi`	sec. 5.9
Utilise un parcours en profondeur lorsque LAR est utilisé	`--lar_dfs`	

TABLE 5.1. – Association des options de `to_parity` à la simplification

5.13.1. Impact du choix entre les algorithmes LAR

Comparaison de CAR et TAR

Nous avons indiqué que lorsque `to_parity` doit appliquer une construction de type LAR, un choix entre CAR, IAR et TAR est effectué en fonction du nombre de couleurs, d'arêtes et de paires de Rabin ou Streett si elles existent. Les options `car`, `iar` et `tar` de `to_parity` permettent de décrire lesquels de ces algorithmes peuvent être utilisés. Il faut mettre en avant que laisser la possibilité de ne pas utiliser CAR et TAR implique que seules les conditions de Rabin et de Streett peuvent être traitées. Utiliser une telle configuration sur un automate ne portant pas une condition de Rabin ou Streett mène à une erreur. De plus, on peut mettre en avant que les simplifications appliquées (si l'option correspondante est activée) peuvent transformer une condition de Rabin en condition quelconque par exemple.

Nous allons d'abord comparer CAR à TAR avant de montrer l'apport du choix entre IAR et CAR par rapport à CAR seul.

Pour la comparaison entre TAR et CAR, la configuration associée à TAR utilise les options

- `tar`
- `bscc`
- `lar_dfs`

indiquant que seul TAR est appliqué et le choix de l'état à traiter s'appuie sur une

Table 5.2. – Répartition des raisons d'échec de TAR et CAR

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
CAR	902	10	28	28	8
TAR	186	0	51	734	5

recherche en profondeur. De plus seule une SCC terminale est conservée. La seconde configuration utilise les options

— `car`
— `lar_dfs`
— `bscc`

et ne diffère que par le remplacement de TAR par CAR.

Pour comparer ces configurations, commençons par nous appuyer sur la table 5.2 qui montre pour les deux configurations pourquoi un résultat n'a pas pu être obtenu. Le premier point à mettre en évidence est le nombre de cas que les deux algorithmes ont été capable de traiter. Parmi les 976 cas testés, l'utilisation de CAR a permis l'obtention de 902 automates de parité contre 186 avec TAR.

Parmi les cas que TAR n'a pas résolu, la cause était un dépassement de mémoire pour 51 cas, un besoin de couleurs trop important (supérieure à 31) pour 734 cas et il y a eu 5 erreurs. Il n'y a cependant eu aucun dépassement de temps.

Parmi les cas que CAR n'a pas résolu, la cause était un dépassement de mémoire pour 28 cas, un besoin de couleurs trop important pour 28 cas, il y a eu 8 erreurs et 10 dépassements de temps.

Il y a eu 2 cas pour lesquels seul TAR a été capable de produire un automate. Cela est dû à un besoin de couleur trop important avec CAR alors que TAR a été capable de terminer. Ces cas viennent tout autant de l'automate produit par `owl` que de l'implémentation de CAR. En effet dans les deux cas les automates portent la condition Inf(19). Cependant CAR va créer une mémoire pouvant porter toutes les couleurs de 0 à 19. L'implémentation place par défaut 19 à droite de la mémoire lors de la construction du premier état. Ainsi voir cette couleur va déplacer les 20 couleurs. Cependant alors que Spot est configuré pour ne supporter que 32 couleurs, ce déplacement implique l'émission de la couleur 39 (en pratique Spot produit un automate de parité maximale impaire où certaines arêtes peuvent ne pas être colorées, ce qui diffère de la définition de CAR donnée).

Considérons maintenant les 184 cas que les deux algorithmes ont en commun et étudions la table 5.3 qui compare la taille des automates obtenus mais aussi la durée de traitement.

Le premier point qui peut être mis en avant est que pour 6 cas, TAR est plus rapide que CAR. À l'inverse, pour 104 cas, CAR a été le plus rapide des deux. Cependant, restreignons-nous maintenant aux 39 cas pour lesquels au moins un des deux algorithmes a eu besoin d'au moins un dixième de seconde. CAR y est alors toujours plus rapide que TAR. Cette différence de vitesse est encore plus évidente à la vue de la durée moyenne

TABLE 5.3. – Comparaison du nombre d'états du DPA et du temps d'exécution avec CAR et TAR sur les 184 cas que les deux algorithmes résolvent. $>_{0,1}$ désigne la restriction aux 39 cas où au moins un des deux algorithmes a besoin d'au moins un dixième de seconde.

	Nombre d'états				Temps (en secondes)					
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)	$>_{0,1}(1)$	$>_{0,1}(2)$
CAR	161	10,30	-	0	0,17	0,07	-	6	-	0
TAR	495414	381,71	176	-	2,35	0,12	104	-	39	-

TABLE 5.4. – Répartition des raisons d'échec de CAR et CAR combiné à IAR

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
CAR	902	10	28	28	8
CAR+ IAR	904	9	26	26	11

de traitement avec TAR où l'on constate qu'en moyenne (géométrique), TAR est presque deux fois plus lent que CAR.

Cette supériorité de CAR se retrouve également dans la taille des automates résultants. En effet, sur les 184 cas étudiés, 176 automates sont plus grands lorsque TAR est utilisé alors que l'inverse ne se produit jamais.

Impact du choix entre CAR et IAR

Nous allons maintenant comparer le cas où CAR est le seul algorithme utilisé et le cas où un choix est fait entre CAR et IAR. Pour rappel, IAR est utilisé si la condition porte moins de paires que de couleurs.

Commençons par étudier la table 5.4 qui montre pour les deux configurations pourquoi un résultat n'a pas pu être obtenu. Ici IAR ne permet pas de résoudre beaucoup plus de cas. En effet, alors que CAR seul peut résoudre 902 des 976 cas, l'ajout de IAR ne permet d'en résoudre que 2 de plus.

Pour les 2 cas où IAR est capable de produire un DPA et pas CAR seul, la cause était un dépassement du nombre de couleurs maximal autorisé. Ces deux cas sont les mêmes que dans la partie précédente et correspondent à des automates portant une unique couleur trop grande pour CAR alors qu'ils ne sont représentés que par une paire avec IAR.

Concentrons-nous alors sur les 277 cas où IAR a été utilisé au moins une fois et où les deux configurations ont donné un résultat et étudions la table 5.5 qui compare la taille des automates obtenus mais aussi la durée de traitement des deux configurations.

Au niveau de la durée d'exécution, il y a 36 cas où CAR seul est la configuration la plus rapide des deux alors que pour 15 cas c'est l'inverse. En se limitant aux 28 cas pour

TABLE 5.5. – Comparaison du nombre d'états du DPA et du temps d'exécution avec CAR et la combinaison de CAR et IAR sur les 277 cas que les deux algorithmes résolvent et où IAR a été appliqué au moins une fois. $>_{0,1}$ désigne la restriction aux 39 cas où au moins un des deux algorithmes a besoin d'au moins un dixième de seconde.

	Nombre d'états				Temps (en secondes)					
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)	$>_{0,1}(1)$	$>_{0,1}(2)$
CAR	13473	78,93	-	106	0,506	0,078	-	15	-	1
CAR+ IAR	13374	70,88	16	-	0,546	0,080	36	-	5	-

TABLE 5.6. – Répartition des raisons d'échec des recherches d'état pour différentes méthodes de recherche d'état existant

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
CAR + IAR	904	9	26	26	11
Recherche plus ancien	902	17	26	26	5
Recherche plus récent	902	25	18	26	5
Recherche a posteriori	899	4	34	26	13

lesquels un des deux algorithmes a eu besoin d'au moins un dixième de seconde, alors CAR seul est plus rapide pour 5 cas alors que c'est l'inverse pour 1 cas.

Au niveau de la taille du résultat, il y a 16 cas où l'utilisation de IAR implique une hausse du nombre d'états alors qu'elle est bénéfique dans 106 cas.

Cette table indique également que lorsqu'il peut être utilisé, IAR permet d'obtenir des automates 1,11 fois plus petits en moyenne (géométrique) par rapport à CAR seul alors que la différence de durée de traitement est négligeable.

5.13.2. Impact de la recherche d'état

La première heuristique que nous allons étudier est la recherche d'état existant. Dans tous les cas l'option indiquant de ne conserver qu'une SCC terminale sera utilisée.

Recherche du plus ancien état compatible

La configuration de base utilise les options

TABLE 5.7. – Comparaison du nombre d'états du DPA et du temps d'exécution avec CAR + IAR et la combinaison de CAR + IAR et de la recherche du plus ancien état compatible sur les 902 cas que les deux algorithmes résolvent.

	Nombre d'états				Temps (en secondes)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
CAR	14486	99	-	495	0,80	0,09	-	56
Rech. ancien	14212	95	28	-	0,99	0,10	170	-

- `bscc`
- `car`
- `iar`
- `lar_dfs`

qui applique CAR ou IAR et ne conserve qu'une SCC terminale. La seconde utilise

- `bscc`
- `car`
- `iar`
- `lar_dfs`
- `search_ex`

qui ajoute à la configuration précédente la recherche du plus ancien état compatible.

Commençons par nous appuyer sur la table 5.6 décrivant entre autres les raisons des échecs pour les deux configurations. On peut y voir que CAR et IAR seuls résolvent 904 des 976 cas et l'utilisation de la recherche d'état implique la résolution de deux cas de moins. Une étude du plus grand cas qui n'est résolu que sans la recherche d'état existant montre que la combinaison de CAR et IAR crée un automate de 9 138 342 états. De plus il n'est composé que de deux SCC dont une portant 512 états. Cela implique donc plusieurs millions d'insertions dans un même arbre dans notre implémentation. Cela explique pourquoi l'utilisation de la recherche d'état existant implique une hausse du nombre de dépassements de temps.

Pour comparer les résultats de ces deux méthodes, appuyons-nous sur la table 5.7 qui compare la taille des automates mais aussi la durée d'exécution pour ces deux méthodes. Sur les 902 cas qu'ils ont en commun et pour lesquels CAR ou IAR a été appliqué, la recherche d'état existant implique la création d'un DPA plus petit pour 495 cas alors que l'inverse se produit pour cas 28 cas.

Si sur le nombre d'états du DPA résultant la recherche d'état existant a un réel bénéfice, il a également un impact sur le temps de traitement. Ainsi pour seulement 56 cas la transformation a été plus rapide en utilisant la recherche d'état existant alors que l'inverse s'est produit pour 170 cas.

TABLE 5.8. – Comparaison du nombre d'états du DPA et du temps d'exécution avec la recherche du plus ancien et du plus récent état compatible sur les 902 cas que les deux algorithmes résolvent.

	Nombre d'états				Temps (en secondes)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
Rech. ancien	14212	95	-	234	0,99	0,10	-	75
Rech. récent	14207	93	0	-	1,02	0,10	100	-

Utilité de la recherche de l'état compatible le plus récent par rapport au plus ancien

Nous décrivons maintenant ce qu'il se passe lorsqu'au lieu de chercher le plus ancien état compatible, on prend le plus récent. Ceci nous donne la configuration

- `bscc`
- `car`
- `iar`
- `lar_dfs`
- `seach_ex`
- `use_last`

Dans la table 5.6, nous pouvons constater que 902 peuvent être résolus lorsque le plus ancien état est recherché, tout comme lorsque l'on cherchait le plus ancien.

Appuyons-nous maintenant sur la comparaison de la taille des automates et des durées de traitement pour ces deux configurations présentées dans la table 5.8. On y voit que la recherche du plus ancien ne produit jamais d'automate plus grand que lorsque le plus ancien est choisi alors que c'est bénéfique dans 234 cas. En moyenne (géométrique), la taille de l'automate est 1,02 plus petite lorsque l'on utilise l'état compatible le plus récent.

Si on s'intéresse maintenant aux temps de traitement, on peut voir qu'il faut plus de temps en recherchant l'état compatible le plus récent pour 100 alors que l'inverse se produit pour 75 cas. Cependant la différence de durée de traitement de ces deux méthodes n'est pas significative.

Utilité de la recherche a posteriori

Maintenant il convient de regarder si rechercher a posteriori améliore encore la taille du résultat et son impact sur la durée de traitement. La configuration utilisée s'appuie sur les options

- `bscc`
- `car`
- `iar`
- `lar_dfs`
- `seach_ex`
- `use_last`
- `use_last_post_process`

TABLE 5.9. – Comparaison du nombre d'état et du temps de traitement entre la recherche de l'état compatible le plus récent et la recherche a posteriori

	Nombre d'états				Temps (en secondes)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
Rech. récent	12105	91	-	369	0,72	0,10	-	82
A posteriori	11188	85	0	-	0,80	0,10	147	-

Cette configuration sera opposée à la recherche du plus récent état compatible déjà décrite.

La table 5.6 nous indique que lorsque la recherche a posteriori est utilisée, 3 de moins sont résolus. On peut remarquer que par rapport aux autres algorithmes, cette recherche a posteriori implique un plus grand nombre de dépassements de mémoire, ce qui diminue le nombre de dépassement de temps.

Appuyons-nous alors sur la table 5.9 pour comparer la taille des automates résultants ainsi que la durée de traitement. Le premier point à mettre en avant est que la recherche a posteriori permet un gain important sur la taille de l'automate puisque le résultat est en moyenne (géométrique) 1,07 fois petit. On peut aussi remarquer que cette recherche a posteriori n'est jamais contre-productive au niveau de la taille du résultat. Sur ces cas qui terminent pour les deux configurations, il n'y a cependant pas de différence significative au niveau de la durée de traitement.

5.13.3. Comparaison de la recherche d'état existant avec l'heuristique sur l'ordre des couleurs

Pour rappel l'heuristique sur l'ordre des couleurs vise à éviter à avoir à chercher un état compatible en attribuant avant le début de la construction un ordre de déplacement pour chaque état et chaque ensemble de couleurs.

Deux données seront étudiées. La première portera sur le nombre d'états de l'automate résultant alors que la seconde décrira le nombre d'états qui ont été créés avant simplification. Ces valeurs peuvent être différentes puisque nous utiliserons l'optimisation consistant à ne conserver qu'une SCC terminale (voir section 5.11.4).

Nous comparerons cette imposition de l'ordre à deux recherches d' état compatible. La première est l'utilisation de l'état le plus récent puisque nous avons vu dans la section 5.13.2 qu'il s'agit d'un compromis intéressant entre le temps de traitement et la taille des automates résultants.

La seconde opposera l'imposition de l'ordre de déplacement et la recherche a posteriori de l'état compatible puisque nous avons vu qu'avec cette dernière le gain en termes de taille était très importants par rapport à la recherche uniquement lors de la construction.

Nous étudierons ensuite l'intérêt de n'imposer l'ordre que lorsqu'aucun état compatible existe.

TABLE 5.10. – Répartition des raisons d'échec avec les recherches d'état et l'imposition de l'ordre

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
Recherche + récent	902	25	18	26	5
Recherche a posteriori	899	4	34	26	13
Imposition ordre	899	28	24	20	5

TABLE 5.11. – Comparaison de la durée de traitement pour les 899 cas que les deux algorithmes résolvent. $>_{0,1}$ désigne la restriction aux 146 cas où au moins un des deux algorithmes a besoin d'au moins un dixième de seconde.

	Tous les cas				Plus d'un dixième seconde			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	$>_{0,1}(1)$	$>_{0,1}(2)$
Recherche + récent	0,72	0,10	-	38	4,13	0,69	-	15
Imposition ordre	0,76	0,10	137	-	4,33	0,72	32	-

La configuration ne s'appuyant que la recherche d'états utilise les options

- `car`
- `iar`
- `search_ex`
- `use_last`
- `lar_dfs`
- `bscc`

alors que celle imposant l'ordre utilise les options

- `car`
- `iar`
- `force_order`
- `lar_dfs`
- `bscc`

Commençons par le nombre de cas résolus. Dans le cas où il y a eu une recherche d'état existant, parmi les 976 cas testés, 902 ont terminé. Dans le cas où l'ordre était imposé, 899 automates ont été obtenus. Pour les cas où l'imposition de l'ordre n'a pas permis de conclure alors la recherche de l'état compatible le plus récent a réussi à terminer, il s'agit d'un manque de mémoire pour deux cas et d'un dépassement de temps pour un cas.

Comparons alors la durée de traitement pour les deux configurations et restreignons-nous aux 902 cas pour lesquels CAR ou IAR a été utilisé (car les automates portant déjà une condition de parité ne sont pas traités par `to_parity`). En étudiant tous les cas, on remarque que l'utilisation de la recherche d'état est plus rapide pour 137 cas alors que c'est le choix de l'ordre qui est plus rapide pour 38 cas. Cependant en se limitant

TABLE 5.12. – Comparaison du nombre d'états du résultat et créés sur les 899 cas que les deux algorithmes résolvent

	Nombre d'états du résultat				Nombre d'états créés			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
Recherche + récent	12105	91	-	387	12356	107	-	669
Imposition ordre	11164	85	0	-	11598	87	30	-

aux cas pour lesquels il faut au moins un dixième de seconde pour au moins une des deux configurations, on se retrouve avec 146 cas. Parmi ces cas, 32 sont favorables à la recherche d'état alors que la contrainte de l'ordre est bénéfique pour 15 cas. Une grande majorité des cas où la recherche est bénéfique en termes de temps est donc constituée des automates rapides à traiter.

Étudions maintenant son impact sur la taille de l'automate résultant. On peut voir qu'il n'existe pas de cas pour lequel la recherche d'état existant donne un meilleur résultat alors que pour 387 cas, c'est l'imposition de l'ordre qui est le meilleur choix. De plus on peut noter qu'il existe 7 cas pour lesquels l'automate produit avec la recherche d'état existant donne un automate deux fois plus grand qu'avec l'imposition de l'ordre. Il s'agit de cas relativement grands puisque les DPA obtenus avec l'imposition de l'ordre ont entre 99 et 54 153 états (245 et 115 844 avec la recherche d'état existant).

Pour ce qui est du nombre d'états créés, il y a en moyenne (géométrique) 107 états créés (91 conservés) avec la recherche d'états existants contre 87 (85 conservés) lorsque l'ordre est imposé.

On peut donc en conclure que l'imposition de l'ordre a un impact très positif sur la taille de l'automate résultant par rapport à la simple recherche de l'état compatible le plus récent, cela a un impact sur la durée de traitement. Cette imposition de l'ordre donne aussi suffisamment d'informations pour ne pas créer beaucoup d'états qui ne seront pas conservés par la suite.

Comparons maintenant l'imposition de l'ordre avec la recherche a posteriori. La configuration associée à cette dernière utilise les options

- `car`
- `iar`
- `search_ex`
- `use_last`
- `use_last_post_process`
- `lar_dfs`
- `bscc`

Puisque nous avons déjà montré que la recherche a posteriori de l'état compatible le plus récent implique un temps de traitement supplémentaire par rapport à la recherche pendant la construction, nous ne nous concentrerons ici que sur le nombre d'états de l'automate final. Le nombre d'états construits ne peut pas changer.

Appuyons-nous sur la table 5.13 comparant la taille des automates obtenus avec ces deux configurations. Sur les 899 cas que les deux configurations ont en commun, il y a 1

TABLE 5.13. – Comparaison du nombre d'états du résultat et créés sur les 899 cas que les deux algorithmes résolvent

	moy. arith.	moy. geom.	>(1)	>(2)
Recherche a posteriori	11188	85	-	73
Imposition ordre	11164	85	1	-

TABLE 5.14. – Répartition des raisons d'échec avec les recherches d'état et l'imposition de l'ordre combinée à la recherche d'état a posteriori

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
Imposition ordre	899	28	24	20	5
Imposition ordre + recherche	897	10	44	20	5

cas pour lequel la recherche a posteriori donne un automate plus petit que lorsque l'ordre est imposé alors que l'inverse est vrai pour 73 cas.

On en conclut donc que l'imposition de l'ordre est une méthode plus efficace que les différentes manières de rechercher un état compatible pour ce qui est de la taille de l'automate résultant même si cela a un impact sur le temps de traitement.

Ajout de la recherche d'état existant à l'imposition de l'ordre

Nous allons maintenant comparer l'imposition de l'ordre avec la combinaison de cette option à une recherche a posteriori. La première configuration utilisera les options

- `bscc`
- `car`
- `force_order`
- `iar`
- `lar_dfs`

déjà décrite précédemment alors que la seconde utilise les options

- `bscc`
- `car`
- `force_order`
- `iar`
- `lar_dfs`
- `search_ex`
- `use_last`
- `use_last_post_process`

dont la diffère se situe dans la recherche d'un état compatible pendant et après la construction.

TABLE 5.15. – Comparaison du nombre d'états du résultat et créés sur les 899 cas que les deux algorithmes résolvent

	Nombre d'états				Durée (s)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
Impos. ordre	9762	83	-	0	0,61	0,10	-	292
Impos. ordre + recherche	9762	83	0	-	0,73	0,10	234	-

En étudiant la table 5.14, on peut voir que parmi les 976 cas considérés, la combinaison des deux options permet d'obtenir 897 DPA, alors que l'imposition de l'ordre seule permet d'obtenir 899 DPA, ce qui est logique puisqu'un travail de recherche d'état supplémentaire est ajouté à l'imposition de l'ordre. On peut également remarquer que la principale différence entre ces deux méthodes est le nombre de dépassements de mémoire, beaucoup plus important qui a mécaniquement réduit le nombre de dépassement de durée.

Sans s'attarder sur une comparaison des durées de traitement, la table 5.15 indique qu'en moyenne (géométrique), la combinaison des options est aussi plus rapide que l'imposition de l'ordre seule sur les 897 cas que les deux configurations peuvent traiter.

On peut également y voir que la combinaison des options n'a eu aucun impact sur la taille des résultats par rapport à l'imposition de l'ordre seule.

5.13.4. Impact de la propagation des couleurs

Impact de la propagation sur LAR

Dans `to_parity`, l'option `propagate_col` contrôlant l'utilisation de la propagation des couleurs est utilisée pour deux cas. Le premier est la simplification de la condition d'acceptation (pour augmenter la probabilité de pouvoir simplifier la condition, voir page 89) et le second vise à donner à CAR et IAR la possibilité de déplacer plus d'éléments lors du traitement d'une arête. Cette option n'influe cependant pas sur la propagation effectuée par la dégénéralisation partielle.

Deux comparaisons seront effectuées. La première comparera CAR seul contre CAR où les couleurs ont été propagées. La seconde comparera CAR à CAR où il y a une propagation utilisée comme une aide pour les simplifications et une autre pour aider CAR à déplacer plus de couleurs.

Dans le détail, la version servant de base utilise les options

- `bscc`
- `car`
- `lar_dfs`
- `search_ex`
- `use_last`
- `use_last_post_process`

TABLE 5.16. – Répartition des raisons d'échec de CAR et CAR combiné à une propagation

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
CAR	897	3	34	28	14
CAR + prop	899	8	26	28	15

TABLE 5.17. – Comparaison du nombre d'états du DPA et du temps d'exécution avec CAR et la combinaison de CAR et la propagation sur les 897 cas que les deux algorithmes résolvent

	Nombre d'états				Nombre d'états créés			Temps (en secondes)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
CAR	11227	88	-	422	113	-	316	0,77	0,10	-	38
CAR + prop	7672	73	29	-	105	305	-	0,83	0,10	222	-

et va donc n'utiliser que CAR pour construire l'automate. À chaque instant sera recherché l'état compatible le plus récent puis une recherche a posteriori sera effectuée. Seule une SCC terminale est conservée.

La seconde configuration utilise les options

- `bscc`
- `car`
- `lar_dfs`
- `propagate_col`
- `search_ex`
- `use_last`
- `use_last_post_process`

qui ne diffère donc de la précédente que par une propagation des couleurs avant l'application de CAR.

Dans tous les cas nous utiliserons une recherche a posteriori d'un état compatible et nous ne garderons qu'une SCC terminale.

Commençons par étudier l'impact de la propagation sur CAR à l'aide de la table 5.16 qui présente les raisons pour lesquelles les deux configurations ont échoué. On peut remarquer qu'alors que CAR seul peut traiter 897 cas, ajouter la propagation permet la résolution de 2 cas supplémentaires. Il s'agit de cas pour lesquels sans la propagation il n'y avait pas suffisamment de mémoire disponible.

Appuyons-nous maintenant sur la table 5.17 qui compare le nombre d'état du résultat ainsi que la durée de traitement pour les deux configurations mais aussi le nombre d'états qui ont été créés durant de traitement. Pour ce qui est de la taille des résultats, l'utilisation de la propagation apporte un réel bénéfice puisque pour 422 cas, cela permet de produire

un automate plus petit que sans. À l'inverse, pour seulement 29 cas l'utilisation de la propagation est contre-productive.

Pour ce qui est du temps de traitement, il y a 222 cas où l'utilisation de la propagation implique un plus grand temps de traitement. Pour 38 cas, le résultat est obtenu plus rapidement avec la propagation. Même si pour une grande partie des cas la propagation permet un gain de taille, en moyenne (géométrique), les automates sont obtenus 1,03 fois plus lentement lorsque la propagation est activée.

On peut également souligner que l'utilisation de la propagation n'implique pas que moins d'états sont créés puisque dans 305 cas l'utilisation de la propagation a impliqué la création de plus d'états alors que l'inverse concerne 316 cas et qu'en moyenne ne pas utiliser la propagation implique la création de 1,083 fois plus d'états qu'avec, ce qui reste négligeable.

Impact de la propagation sur la simplification d'automate

Nous allons maintenant voir si appliquer une propagation avant la simplification et pour CAR permet d'améliorer les résultats par rapport à une application de CAR sans propagation mais avec une simplification.

La configuration de base utilise les options

- `acc_clean`
- `bscc`
- `car`
- `lar_dfs`
- `search_ex`
- `use_last`
- `use_last_post_process`

qui applique CAR en recherchant l'état le plus récent pendant et après la construction. Seule une SCC terminale est conservée et une simplification de la condition est effectuée.

La seconde configuration utilise les options

- `acc_clean`
- `bscc`
- `car`
- `lar_dfs`
- `propagate_col`
- `search_ex`
- `use_last`
- `use_last_post_process`

qui ne diffère de la précédente que par une propagation avant la simplification et une autre pour aider CAR.

Commençons par étudier la table 5.18 décrivant les raisons des échecs pour les deux configurations. La première chose à noter est qu'alors qu'en utilisant uniquement la simplification, 908 cas sont traités, alors que l'ajout de la propagation permet d'en résoudre 913.

Pour 4 des cas où ils diffèrent, il s'agit d'un dépassement de mémoire lorsque la propagation n'est pas utilisée alors qu'il s'agit d'un dépassement de temps pour un cas.

Appuyons-nous maintenant sur la table 5.19 comparant la taille des résultats ainsi que la durée de traitement pour ces deux configurations. Une amélioration du nombre

TABLE 5.18. – Répartition des raisons d'échec de CAR avec une simplification et CAR avec une simplification combinée à une propagation

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
CAR + simpl.	908	11	27	14	16
CAR + simpl. + prop.	913	13	18	14	18

TABLE 5.19. – Comparaison du nombre d'états du DPA et du temps d'exécution avec CAR avec une simplification et la combinaison de CAR, une simplification et une propagation sur les 908 cas qu'ils ont en commun

	Nombre d'états				Temps (en secondes)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
CAR + simpl.	19372	94	-	452	1,24	0,11	-	411
CAR + simpl. + prop.	7331	72	4	-	0,68	0,10	204	-

d'états est aussi visible puisque pour 452 cas, une propagation avant la simplification réduit le nombre d'états contre 4 cas pour lesquels l'inverse est constaté. De plus les automates résultants sont en moyenne 1,30 fois plus grands lorsque la simplification n'est pas précédée d'une propagation.

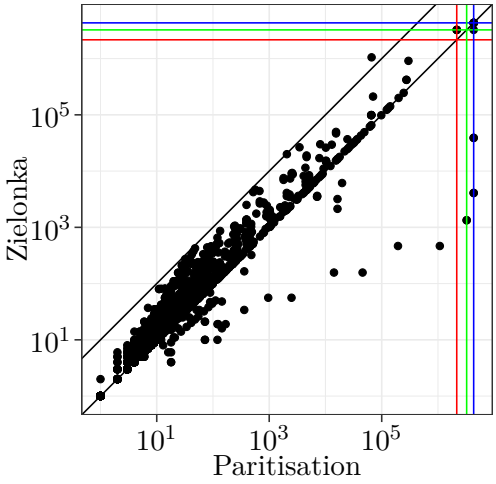
La durée de traitement incite également à propager les couleurs avant une simplification puisque cela est bénéfique pour 411 des 913 cas où LAR a été utilisé. L'inverse se produit pour 204 seul cas. En moyenne la différence des durées de traitement reste négligeable puisque le rapport moyen (géométrique) entre le temps avec et sans cette propagation est de 1,10.

5.13.5. Proximité de LAR optimisé avec les arbres de Zielonka

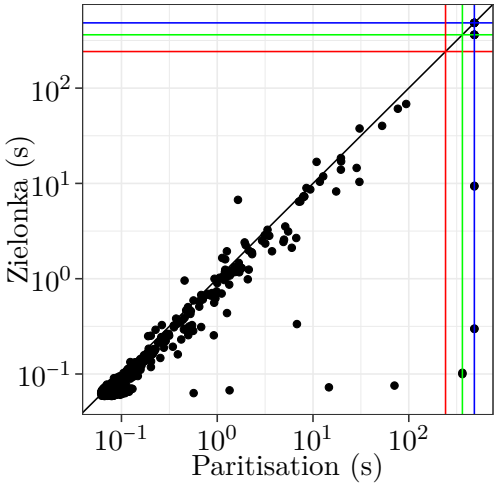
Pour rappel, l'utilisation d'un arbre de Zielonka permet de construire un automate de parité minimal reconnaissant une condition. Ainsi, quelle que soit la version de LAR utilisée, il n'est pas possible d'obtenir un automate plus petit que celui obtenu à l'aide d'un arbre de Zielonka.

Cependant, l'utilisation de diverses optimisations telles que la dégénéralisation partielle ou la simplification de condition par exemple peuvent aider LAR à produire un automate plus petit qu'avec les arbres de Zielonka. Nous allons voir ici si ces optimisations permettent de repousser les limites des arbres de Zielonka.

Nous allons ici comparer l'utilisation des arbres de Zielonka (associée à la fonction



(a) Comparaison du nombre d'états. La ligne parallèle à la diagonale centrale représente les cas pour lesquels `to_parity` produit des automates dix fois plus petit qu'avec les arbres de Zielonka



(b) Comparaison de la durée de traitement

FIGURE 5.25. – Comparaison de `to_parity` et des arbres de Zielonka. Les lignes rouges correspondent aux dépassements de temps, les vertes au manque de mémoire, les bleues aux autres problèmes tels que le nombre de couleurs ou l'impossibilité d'allouer la mémoire.

`zielonka_tree_transform` de Spot) à l'utilisation de `to_parity` où les options suivantes sont utilisées :

- `acc_clean`
- `bscc`
- `car`
- `force_order`
- `iar`
- `lar_dfs`
- `parity_equiv`
- `parity_prefix`
- `partial_degen`
- `propagate_col`
- `rabin_to_buchi`

Dans la figure 5.25, nous pouvons voir que sur les 976 cas traités, `to_parity` a été capable de produire 961 DPA alors qu'avec les arbres de Zielonka, 967 ont été obtenus.

Au niveau de la durée de traitement, il y a 14 cas où `to_parity` a été le plus rapide des deux alors que l'inverse a eu lieu pour 552 cas.

Même si l'utilisation d'arbre de Zielonka est clairement avantageuse au niveau de la durée d'exécution, comparons maintenant la taille des DPA produits. On peut commencer par remarquer que pour 818 cas, l'automate obtenu avec `to_parity` est plus petit que celui obtenu à l'aide des arbres de Zielonka alors que l'inverse ne concerne que 54 cas.

On peut également souligner que pour 311 cas, `to_parity` produit un automate au moins 2 fois plus petit qu'avec les arbres de Zielonka alors que l'inverse se produit 34 fois.

5.14. Conclusion

Nous avons ici présenté différentes méthodes de paritisation et les avons regroupés en une unique procédure `to_parity` tout en y ajoutant diverses optimisations. Nous avons vu qu'en pratique elle reste compétitive face à l'utilisation d'arbres de Zielonka. Même si ces derniers assurent une forme d'optimalité par rapport aux algorithmes de la famille LAR, nous avons vu que les diverses optimisations introduites permettent en pratique à `to_parity` d'obtenir de meilleurs résultats.

6. Alternating Cycle Decomposition

Le travail que nous allons décrire dans ce chapitre est à lier à l'article « Practical Applications of the Alternating Cycle Decomposition » [15] présenté à TACAS'22.

Tout comme dans le chapitre 5 le but est de transformer un automate à condition quelconque en un automate de parité. Cependant nous nous appuyons ici sur une procédure plus récente nommée ACD [14] garantissant une certaine optimalité sur la taille et le nombre de couleurs de l'automate résultant.

Alors que Casares, Colcombet et Fijalkow [14] en firent une description pour les automates portant une condition de Muller, nous allons l'adapter au cadre de conditions d'Emerson-Lei. Nous allons également voir si en pratique cet algorithme est utilisable en le comparant avec la procédure `to_parity`, puisqu'avec Spot et Owl nous disposons des deux premières implémentations.

6.1. Description de ACD

ACD (pour Alternating Cycle Decomposition) est un algorithme permettant de produire un automate de parité à partir d'un automate portant une condition quelconque. Il s'agit d'une amélioration des arbres de Zielonka. Tout comme pour cette dernière il s'agit de créer un arbre et de créer un automate de parité où chaque nœud porte un état de l'automate de départ et une feuille de l'arbre. Cependant ici nous aurons un arbre par SCC et chaque nœud portera une SCC qui n'est pas forcément maximale.

Dans ce chapitre notre objectif est de transformer un automate d'Emerson-Lei $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ en un automate de parité $\mathcal{P}^{\mathrm{ACD}}$ qui lui est équivalent.

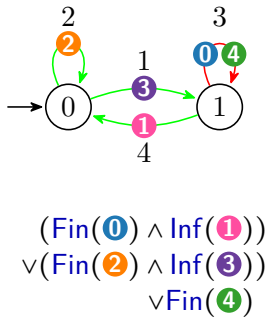
6.1.1. Forêt ACD

À la différence des arbres de Zielonka, ACD va s'appuyer sur un ensemble d'arbres. Dans ces arbres, chaque nœud représente un ensemble de cycles non disjoints, c'est-à-dire une SCC qui n'est pas forcément maximale. (alors qu'il s'agit d'un ensemble de couleurs pour les arbres de Zielonka) et son acceptation dépend de l'acceptation de la SCC qu'il porte. L'idée d'une alternance de nœuds acceptants et rejetants est conservée.

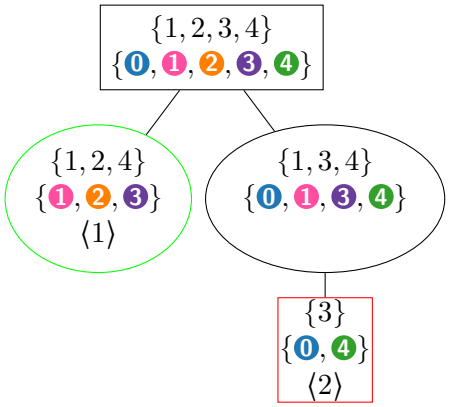
De manière plus formelle, l'*ACD* de $\mathcal{A}$ est définie de la manière suivante :

Définition 68 (ACD)**.** *Soit $S_1, \ldots, S_k$ une énumération des SCC de $\mathcal{A}$. L'ACD de $\mathcal{A}$, notée $\mathcal{ACD}(\mathcal{A})$, est un ensemble de k arbres étiquetés par des cycles de $\mathcal{A}$ $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ où $\mathcal{T}_i = (T_i, \eta_i) \in 2^Q \times 2^E$ tel que :*

— *la racine v_0 de T_i est telle que $\eta_i(v_0)$ correspond à l'ensemble des arêtes de S_i ;*



(a) Automate de Rabin. À chaque arête est associé un identifiant



(b) ACD associé à l'automate de la figure 6.1a

FIGURE 6.1. – Exemple d'ACD

— *si* $x \in T_i$ *et* $\eta_i(x)$ *est un cycle acceptant, alors pour chaque élément maximal de l'ensemble* $\{\ell \in \mathit{cycle}(\mathcal{A}) \mid \ell \subseteq \eta_i(x)$ *et* ℓ *est rejetant}, x *a un fils*

— *si* $x \in T_i$ *et* $\eta_i(x)$ *est un cycle rejetant, alors pour chaque élément maximal de l'ensemble* $\{\ell \in \mathit{cycle}(\mathcal{A}) \mid \ell \subseteq \eta_i(x)$ *et* ℓ *est acceptant}, x *a un fils*

Comme pour les arbres de Zielonka, les nœuds *portant un cycle acceptant seront des nœuds ovales et les autres seront des nœuds rectangulaires.*

Pour un état q *correspondant à la SCC* S_i *de* $\mathcal{A}$*, on définit l'arbre associé à* q *comme* $\mathcal{T}_q = (T_q, \eta_q)$ *tel que*

$$T_q = \{x \in T_i \,:\, q \in \mathit{states}(\eta_i(x))\}\,, \quad \eta_q = \eta_i|_{T_q}.$$

Exemple 20 (Construction d'un ACD)**.** *Illustrons la construction d'un ACD sur l'automate de la figure 6.1a. Puisque cet automate n'a qu'une SCC, nous n'allons avoir qu'un arbre.*

La première étape consiste à étudier l'ensemble des arêtes de la SCC. Le cycle passant par ces arêtes porte l'ensemble de couleurs $\{0, 1, 2, 3, 4\}$ *qui est rejetant pour la condition de l'automate.*

Ainsi la racine de l'ACD de la figure 6.1b porte ce cycle et est acceptante. On y représente l'ensemble des couleurs pour une question de lisibilité mais ce dernier peut être construit à partir de l'ensemble d'arêtes.

Il s'agit ensuite de trouver les cycles acceptants maximaux contenus dans le cycle de la racine. Ces deux cycles portent les arêtes $\{1, 2, 4\}$ *et* $\{1, 3, 4\}$ *et sont donc associés aux deux fils de la racine.*

6.1.2. Construction d'un automate de parité à partir d'un ACD

De manière analogue à la transformation basée sur un arbre de Zielonka, nous allons ici nous appuyer sur un parcours de l'ACD.

Pour décrire cela, introduisons certains mouvements dans un tel arbre.

Étant donnée une arête $e = q \xrightarrow{\ell,c} q'$ telle que q et q' appartiennent à la i^{e} SCC de $\mathcal{A}$ et $x \in T_i$, on définit $\mathit{Support}(x,e)$ comme étant le plus bas ascendant z de x dans T_i tel que $e \in \eta_i(z)$. Si $\mathit{Support}(x,e) \neq x$ et n'est pas une feuille de $\mathcal{T}_{q'}$, notons z' l'unique enfant de $\mathit{Support}(x,e)$ qui est un ascendant de x et soit $y_1, \dots, y_s$ une énumération de gauche à droite des nœuds de $\text{Children}_{\mathcal{T}_{q'}}(\mathit{Support}(x,e))$. On définit alors $\mathit{NextBranch}(x,e)$ comme :

$$\begin{cases} \mathit{Support}(x,e) & \text{si } \mathit{Support}(x,e) = x \text{ ou si } \mathit{Support}(x,e) \text{ est une feuille de } T_{q'} \text{ ;} \\ y_1 & \text{si } z' = y_s \text{ ;} \\ y_{j+1} & \text{si } z' = y_j, 1 \leq j < s. \end{cases}$$

On peut alors définir l'automate de parité $\mathcal{P}^{\text{ACD}} = (Q^{\text{ACD}}, M', \Sigma, \delta', q'_0, \beta)$ de la manière suivante :

États Les états de $\mathcal{P}^{\text{ACD}}$ sont de la forme (q,x) où $q \in Q$ et x est une feuille de l'arbre associé à q. L'état initial est de la forme (q_0, x) où x est la feuille la plus à gauche de l'arbre associé à q_0.

Arêtes Pour chaque arête $e = q \xrightarrow{\ell,c} q' \in \delta$ et chaque état $(q,x) \in Q^{\text{ACD}}$, définissons l'arête $(q,x) \xrightarrow{\ell,c'} (q',y) \in \delta'$ de la manière suivante : Si q et q' n'appartiennent pas à la même SCC, y est la feuille la plus à gauche de $T_{q'}$ et $c' = 1$ (sauf si tous les T_i ont une hauteur de 1 et une racine ovale, dans ce cas $c' = 0$). Sinon, la feuille y est le descendant le plus à gauche de $\mathit{NextBranch}(x,e)$ dans $T_{q'}$. La couleur c' de l'arête est $\mathit{Depth}(\mathit{Support}(x,e))$ si la racine r de T_i est un nœud ovale ($\eta_i(r) \vDash \alpha$) ou $\mathit{Depth}(\mathit{Support}(x,e)) + 1$ sinon.

Condition de parité La condition de parité β est une condition de parité minimale paire.

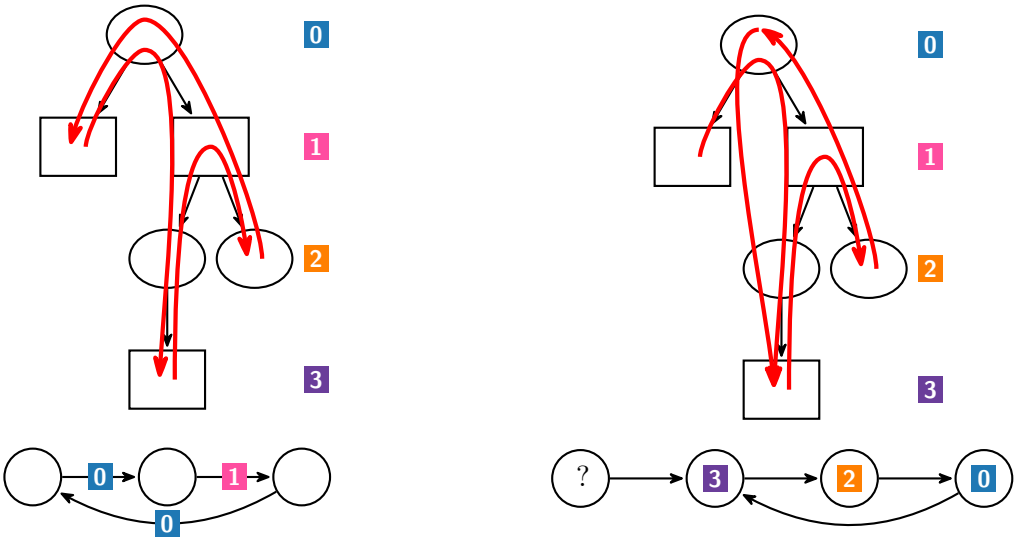
Remarque 52. *Si* 0 *n'apparaît sur aucune arête, on peut décaler toutes les couleurs de -1 et remplacer β par une condition de parité minimale impaire.*

Tout comme les automates produits par CAR, IAR..., $\mathcal{P}^{\text{ACD}}$ reconnaît $\mathcal{L}(\mathcal{A})$ et il préserve de nombreuses propriétés de l'automate de départ :

— sa complétude ;

— son déterminisme ;

— son ambiguïté, c'est-à-dire s'il existe pour chaque mot au plus une exécution acceptante ;



(a) Cas où les arêtes sont colorées

(b) Cas où les états sont colorés

FIGURE 6.2. – Différence entre la coloration des arêtes et des états avec ACD.

— être Good for games [35], c'est-à-dire un automate dont le non-déterminisme peut être résolu par une analyse du préfixe déjà lu.

Remarque 53. *Par construction de l'arbre chaque nœud a au moins une couleur de moins que son parent et le nombre de couleurs de $\mathcal{P}^{ACD}$ est majoré par la hauteur des arbres de l'ACD associé.*

Ainsi l'automate de parité obtenu porte au plus $|M|+1$ couleurs et puisque la définition de TELA n'impose pas une coloration de toutes les arêtes, on peut limiter cette borne à $|M|$ couleurs en ne coloriant pas le dernier niveau de la forêt.

6.2. Adaptation de la transformation pour le cas où les états sont colorés

Nous allons maintenant décrire comment ACD peut être utilisé pour obtenir des automates de parité où les couleurs sont portées par les états.

Pour comprendre l'idée de la construction, appuyons-nous sur la figure 6.2 qui montre la construction d'un cycle à partir d'un arbre ACD pour les deux méthodes de coloration. D'un côté, on voit sur la figure 6.2a le cas où les arêtes sont colorées. On peut remarquer que pour ce cycle, la seule couleur qui nous intéresse est 0 puisqu'il s'agit de la plus petite couleur vue dans ce cycle. De manière générale, lorsqu'un cycle est construit avec ACD, la seule couleur importante est celle correspondant au plus haut niveau vu. C'est-à-dire, le niveau utilisé lors du retour en arrière. Pour adapter la construction au cas où les états sont colorés, on va donc procéder ainsi : lorsque l'on va se déplacer vers la droite dans l'arbre, la couleur associée à la destination sera produite, alors que lors du retour

en arrière, ce sera la couleur du plus haut niveau traversé. Cette méthode est décrite dans la figure 6.2b.

Définissons maintenant de manière plus formelle la construction de cet automate. Un automate dont les états sont colorés est de la forme $\mathcal{A} = (Q, M, \Sigma, q_0, \delta, \Gamma)$ où $(Q, M, \Sigma, q_0, \delta)$ est similaire à la définition habituelle sauf $\delta \subseteq Q \times \Sigma \times Q$, et $\Gamma : Q \to M$ associe à chaque état une couleur.

Soient $\mathcal{A}$ un TELA et $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \dots, \mathcal{T}_k \rangle$.

On définit l'automate de parité sur les états $\mathcal{P}^{\text{sb-ACD}} = (Q^{sb-ACD}, M', \Sigma, \delta', q_0', \Gamma')$ de la manière suivante :

États Les états sont de la forme (x, q) pour $q \in Q$ et $x \in T_q$ (x n'est donc pas forcément une feuille ici).

$$Q^{\text{sb-acd}} = \bigcup_{q \in Q} \{q\} \times T_q \ , \quad q_0 = (q_0, x). \text{ où } x \text{ est la feuille la plus à gauche de } \mathcal{T}_{q_0}$$

Arêtes Pour toute transition $e = q \xrightarrow{\ell, C} q' \in \Delta$ et $(q, x) \in Q^{\text{sb-ACD}}$, on a $(q, x) \xrightarrow{\ell} (q', y)$. La valeur de y est calculée ainsi :

— Supposons que x est une feuille de $\mathcal{T}_q$. Si $\mathit{NextBranch}(x, e)$ n'est pas l'enfant le plus à gauche de $\mathit{Support}(x, e)$ dans $\mathcal{T}_{q'}$, alors y est la feuille la plus à gauche en dessous de $\mathit{NextBranch}(x, e)$ dans $\mathcal{T}_{q'}$. Si $\mathit{NextBranch}(x, e)$ est l'enfant le plus à gauche, alors $y = \mathit{Support}(x, e)$.

— Si x n'est pas une feuille de $\mathcal{T}_q$, la destination y est calculée comme si la transition partait de (q, x') où x' est la feuille la plus à gauche sous q.

Condition de parité $\Gamma((q, x)) = \mathit{Depth}(x)$ si la racine de $\mathcal{T}_q$ est un nœud rond, et $\Gamma((q, x)) = \mathit{Depth}(x) + 1$ sinon.

À la différence du cas où les arêtes sont colorées, nous n'avons ici aucune garantie d'optimalité du résultat. En effet, si x n'est pas une feuille, alors les états de la forme $(q, x) \in Q^{\text{sb-ACD}}$ ne sont pas forcément accessibles dans $\mathcal{P}^{\text{sb-acd}}$. Nous n'avons besoin que d'ajouter ceux qui peuvent être atteint depuis l'état initial. Cependant l'ensemble des états accessibles dépend de l'ordonnancement des enfants dans les arbres de l'ACD et la taille du résultat dépend donc de cet ordre.

Nous utilisons l'heuristique suivante. Pour un arbre $\mathcal{T}_i$ et $x \in T_i$, on définit

$$D_i(x) = \{q' \in Q \mid q \xrightarrow{a} q \notin \eta_i(x) \text{ pour } q \in \mathit{states}(\eta_i(x)) \text{ et } a \in \Sigma\}$$

L'heuristique consiste alors à trier les fils d'un nœud par $|D_i(x)|$ décroissant.

6.3. Typage d'automate

Nous avons défini dans le chapitre 5 la notion de typage d'automate (Büchi-type, parity-type).

Nous allons ici nous appuyer sur l'ACD pour déterminer si un automate fait partie de ces classes d'automates.

Proposition 2 ([14]). *Soient $\mathcal{A}$ un TELA déterministe tel que tous ses états $q \in Q$ sont accessibles et $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ sa décomposition ACD. On a alors :*

— *$\mathcal{A}$ est Rabin-type (resp. Streett-type) si et seulement si pour tout état $q \in Q$, tout état rond (resp. rectangulaire) de $\mathcal{T}_q$ a au plus un enfant dans $\mathcal{T}_q$. Il est parity-type s'il est à la fois Rabin-type et Streett-type.*

Puisque nous travaillons sur des TELA, nous pouvons étendre ce théorème aux automate generalized Büchi-type :

Proposition 3 ([15]). *Soient $\mathcal{A}$ un TELA déterministe tel que tous ses états $q \in Q$ sont accessibles et $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ sa décomposition ACD. On a alors :*

— *$\mathcal{A}$ est generalized Büchi-type (resp. generalized co-Büchi-type) si et seulement si pour tout $1 \leq i \leq k$, $Height(\mathcal{T}_i) \leq 2$ et en cas d'égalité, la racine de T_i est un nœud rond (resp. rectangulaire) ;*

— *$\mathcal{A}$ est faible si et seulement si pour tout $1 \leq i \leq k$, $Height(\mathcal{T}_i) = 1$.*

Remarque 54. *L'ACD donne un typage pour chaque SCC de l'automate, ce qui permet également de simplifier la condition d'acceptation pour chacune d'entre elles indépendamment. De plus, les implications de droite à gauche dans la proposition 3 restent vraies dans le cas non-déterministe.*

6.4. Étude de ACD avec Spot et Owl

Nous allons comparer ici quatre transformations. La première est l'utilisation d'arbres de Zielonka et la deuxième est la procédure associée à `to_parity`. Les deux autres sont les implémentations de ACD de Spot et Owl. Il s'agit d'une comparaison que nous avons décrite durant la conférence TACAS 2022 [15]. Cependant depuis cette publication une mémoïsation a été ajoutée à la construction des arbres de Zielonka. De plus `to_parity` a été réécrit, ce qui a permis une amélioration du temps de traitement ainsi que l'introduction d'optimisation qui n'était pas présentes dans la version présentée en 2020 [65].

6.4.1. Comparaison entre ACD et `to_parity`

Nous allons tout d'abord comparer ACD avec `to_parity`. Pour ce dernier, les options utilisées sont :

— `acc_clean`
— `bscc`
— `car`
— `force_order`
— `iar`
— `lar_dfs`
— `parity_equiv`
— `parity_prefix`
— `partial_degen`
— `propagate_col`
— `rabin_to_buchi`

TABLE 6.1. – Répartition des raisons d'échec de ACD et `to_parity`

	# Cas résolus	# dépassements de temps	# dépassements de mémoire	# dépassements de couleur	# erreurs
ACD	966	2	3	0	5
`to_parity`	961	3	0	5	7

TABLE 6.2. – Comparaison du nombre d'états du DPA avec ACD et `to_parity` sur les 960 cas que les deux algorithmes résolvent

	Nombre d'états			
	moy. arith.	moy. geom.	>(1)	>(2)
ACD	2886	43	-	0
`to_parity`	4419	51	113	-

Comme dans le chapitre 5, la limite de temps est fixée à 120 secondes et la mémoire disponible à 6000 Mo.

Commençons par étudier le nombre de cas que les deux algorithmes ont traité.

Pour cela, appuyons-nous sur la figure 6.1. Cette dernière décrit la cause des échecs (dépassement de temps, de mémoire, erreur) de ces deux procédures.

La première remarque qui peut être faite concerne le nombre de cas traités par les deux algorithmes. Sur les 976 automates fournis, 966 ont été paritisés avec ACD alors que `to_parity` en a traité 961.

Sur les 6 cas que seul ACD peut traiter, `to_parity` a échoué 4 fois à cause de la production d'une trop grande couleur (limitée à 31) alors que pour les deux autres, il s'agit d'une allocation de mémoire ayant échoué. À l'inverse, il existe un cas pour lequel seul `to_parity` a été capable de terminer. Il s'agit d'un automate à 43 009 états et 3 402 754 arêtes dont la condition est $(\bigwedge_{0\leq i\leq 12} \mathsf{Inf}(i)) \vee \mathsf{Fin}(13)$. Alors que pour traiter cet automate ACD va créer un arbre avec 13 329 nœuds, `to_parity` va appliquer une dégénéralisation partielle, ce qui donne la condition $\mathsf{Inf}(0) \vee \mathsf{Fin}(13)$ qui est une condition de parité maximale impaire si on remplace 0 par 1 et 13 par 0.

Étudions maintenant la taille des automates résultants à l'aide de la table 6.2.

On peut y remarquer que lorsque les deux algorithmes terminent, ACD ne produit pas d'automate plus grand qu'avec `to_parity`, ce qui vérifie bien son optimalité.

Sur les 960 cas traités par les deux algorithmes, il y en a 113 pour lesquels `to_parity` donne un plus grand automate qu'avec ACD.

Cependant une étude des 183 cas pour lesquels `to_parity` a eu besoin d'appliquer LAR montre qu'il n'a été capable de donner un résultat optimal que pour 82 cas, ce qui montre que malgré nos optimisations, il reste difficile d'obtenir un résultat optimal avec les algorithmes de la famille LAR.

Nous avons donc vu que ACD arrive à résoudre plus de cas que `to_parity`, et que ce dernier ne donne pas de résultat optimal pour environ 44% des cas. Voyons maintenant si

TABLE 6.3. – Comparaison de la durée de traitement avec ACD et `to_parity` sur les 960 cas que les deux algorithmes résolvent

	Tous les cas (en sec.)				Plus d'un dixième de seconde (en sec.)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
ACD	0,55	0,09	-	15	5,10	0,46	-	15
`to_parity`	0,71	0,10	443	-	6,93	0,66	133	-

TABLE 6.4. – Comparaison du nombre d'états du DPA avec ACD et les arbres de Zielonka sur les 966 cas que les deux algorithmes résolvent

	Nombre d'états			
	moy. arith.	moy. geom.	>(1)	>(2)
ACD	2897	44	-	0
Zielonka	5533	83	876	-

la qualité supérieure des automates produits par ACD est contrebalancée par une durée de traitement plus importante.

Pour cela, nous nous appuyons sur la table 6.3. On y voit que sur l'ensemble des cas, il n'y a pas de très grand écart de temps. Cependant, pour les 182 cas où au moins un des deux algorithmes a eu besoin d'au moins un dixième de seconde, alors `to_parity` est en moyenne 1,44 fois plus lent que ACD.

Parmi les cas où la durée de traitement des deux algorithmes diffère significativement, on peut extraire un ensemble de cas que ACD est capable de traiter en moins d'un dixième de seconde alors que `to_parity` a besoin d'au moins dix fois plus de temps. Il s'agit des cas de la famille `ltl2dba_RX`. Pour `ltl2dba_R5` par exemple, il s'agit d'un automate à un état et 64 arêtes dont la condition porte sur 10 couleurs. Sur cette condition à 11 couleurs, `to_parity` applique CAR et un automate portant 45 710 états est créé. Pour ACD, l'arbre va uniquement contenir 171 nœuds et l'automate résultant aura 62 états.

On peut donc conclure que lors du traitement d'automates liés à la synthèse LTL, l'écart de taille des automates de parité produits par ACD et `to_parity` reste très faible.

La durée de traitement penche en faveur de ACD même si nous avons vu qu'il existe un automate à 43 009 états que `to_parity` va traiter beaucoup plus rapidement que ACD en ne s'appuyant que sur une dégénéralisation partielle pour obtenir une condition équivalente à de la parité.

6.4.2. Comparaison d'ACD avec les arbres de Zielonka

Nous avons vu en section 6.4.1 qu'en pratique ACD est plus rapide que `to_parity` mais aussi dans la section 5.13.5 que l'utilisation des arbres de Zielonka était également plus rapide que `to_parity`. Nous allons alors ici comparer ACD avec l'utilisation des arbres de Zielonka.

Commençons par une étude rapide du nombre d'états des automates de parité produits

TABLE 6.5. – Comparaison de la durée de traitement avec ACD et les arbres de Zielonka sur les 966 cas que les deux algorithmes résolvent

	Tous les cas (en sec.)				Plus d'un dixième de seconde (en sec.)			
	moy. arith.	moy. geom.	>(1)	>(2)	moy. arith.	moy. geom.	>(1)	>(2)
ACD	0,56	0,09	-	165	5,98	0,62	-	35
Zielonka	0,50	0,09	66	-	3,29	0,58	44	-

à travers la table 6.4. Il s'agit là encore des cas qui peuvent être traités en moins de 120 secondes avec au plus 6Go de RAM. Sur les 976 cas, on peut voir qu'avec les arbres de Zielonka, le résultat n'a été optimal que pour 90 d'entre eux. De plus, les automates obtenus à l'aide d'arbres de Zielonka sont en moyenne (géométrique) 1,86 fois plus grand que le résultat optimal.

Intéressons-nous alors aux temps de traitement pour ces deux algorithmes. Pour cela, servons-nous de la table 6.5. Pour rappel, un algorithme A est considéré comme plus rapide qu'un algorithme B si la durée avec A multipliée par 1,1 est inférieure à la durée avec B.

On y voit une grande proximité des temps de traitement. Cependant pour les cas où au moins un des deux algorithmes a besoin d'au moins un dixième de seconde, la durée de traitement est en moyenne 1,07 fois plus rapide avec les arbres de Zielonka.

6.4.3. Comparaison des implémentations d'ACD

Nous allons maintenant comparer les deux implémentations de ACD qui ont été faites dans le cadre de « Practical Applications of the Alternating Cycle Decomposition » [15]. La première est dans l'outil Owl alors que la seconde est celle de Spot utilisée depuis le début de cette section. Puisque ACD assure une optimalité du résultat la taille des automates résultants ne sera pas comparée ici. Il s'agira uniquement d'une comparaison du temps de traitement. Cette étude permettra également de mettre en évidence les différences entre les deux implémentations. Au-delà de la différence d'implémentation de ACD dans les deux outils, il existe également des différences dans les autres parties des outils. Ainsi avec Owl, la lecture de l'automate est relativement plus longue qu'avec Spot alors que l'initialisation de l'outil est plus longue dans Spot. Au lieu de mesurer le temps de traitement total des outils, nous n'utilisons que la durée de transformation. La limite de temps est de 500 secondes et la limite pour la mémoire est de 5120 Mo mais s'applique à l'outil entier.

Ici nous allons restreindre notre ensemble de 1462 machines déjà utilisées précédemment aux 1065 cas qui portent au moins deux couleurs. Parmi ces cas, 93 portent déjà une condition de parité et sont représentés en bleu.

Pour cette comparaison, nous nous appuyons sur la figure 6.3. Les durées de paritisation des deux implémentations y sont présentées et une coloration des points permet de montrer les cas portent déjà une condition de parité.

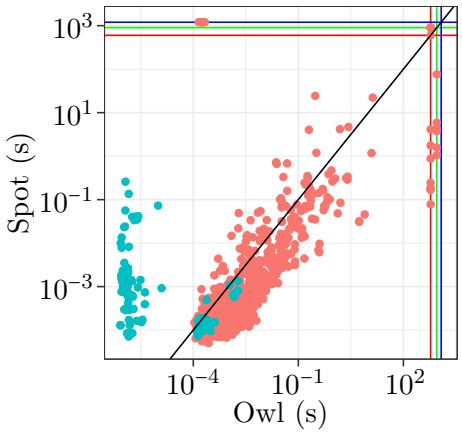


FIGURE 6.3. – Comparaison de la durée d'exécution de ACD avec Spot et Owl. Les points bleus correspondent aux automates portant déjà une condition de parité. Les lignes rouges indiquent les dépassements de temps, les vertes un manque de mémoire et les bleues un manque de couleur.

Remarque 55. *Les temps mesurés pour la version de ACD dans Spot ne sont pas les mêmes que dans les parties précédentes puisqu'il a fallu modifier le protocole pour ne mesurer que le temps de transformation afin que les deux outils soient traités équitablement.*

Le premier point à mettre en avant est qu'avec la version de Owl 1046 cas ont été traités alors que la version de Spot en a traité 1055. Dans le détail, Owl a échoué 6 fois à cause d'un manque de mémoire et 13 fois à cause d'un dépassement de temps. Pour Spot, il s'agit d'un manque de mémoire pour 4 cas, alors qu'un seul dépassement de temps existe. On peut également mettre en avant que pour 5 cas une couleur trop grande est produite par Spot (ce qui n'est pas possible avec Owl).

Si on regarde plus en détail les cas où seul un des outils a donné un résultat, on peut constater que dans les cinq cas pour lesquels seul Owl a été capable de terminer la raison de l'échec est la production d'une trop grande couleur. Cependant il s'agit uniquement de cas pour lesquels c'est l'automate de départ qui est trop grand. En effet ce ne sont que des automates portant la condition $\mathsf{Inf}(54)$ qui est donc équivalente à une condition de Büchi.

Si on s'intéresse maintenant aux 14 cas où seule la version de Spot a terminé, alors on constate qu'il s'agit d'un manque de mémoire pour six cas alors que le reste est un dépassement de temps.

Remarque 56. *La consommation de RAM supplémentaire de Owl est en moyenne (géométrique) 2,30 fois plus importante que Spot (46.86 Mo contre 20,37 Mo).*

Comparons tout d'abord les 948 cas (en rouge) où l'automate ne portait pas déjà une condition de parité. Nous considérons la même définition d'algorithme plus rapide qu'un autre que celle utilisée précédemment (le facteur 1,1). Nous obtenons alors 87 cas où la version de Owl a été la plus rapide contre 839 pour celle de Spot. Si on se limite aux 30

cas pour lesquels au moins un des outils a eu besoin d'au moins une demie seconde, alors Owl a été le plus rapide dans 9 cas contre 21 pour Spot. De plus, Spot a été en moyenne (géométrique) 2,67 fois plus rapide (0,94 secondes conte 0,35).

Revenons maintenant aux cas qui portent déjà une condition de parité. On peut remarquer que pour quasiment tous ces cas le traitement effectué en moins de deux centièmes de secondes par Owl alors que Spot peut avoir besoin d'un quart de seconde. Cette différence s'explique par le choix dans Spot de calculer l'ACD même lorsque l'automate porte déjà une condition de parité. Cela permet de minimiser le nombre de couleurs de l'automate. Owl qui cherche juste à obtenir un DPA n'a pas fait ce choix, ce qui explique cette différence. On peut cependant noter qu'il existe un ensemble de cas où l'automate donné porte déjà une condition de parité mais pour lesquels Owl a besoin d'autant de temps que Spot. Il s'agit de condition de Streett et Rabin à une paire, c'est-à-dire $\mathsf{Fin}(\mathbf{0}) \wedge \mathsf{Inf}(\mathbf{1})$ et $\mathsf{Fin}(\mathbf{0}) \vee \mathsf{Inf}(\mathbf{1})$ qui ne sont pas vues comme des conditions de parité par Owl, et ACD est donc bien appliqué par cet outil.

6.5. Conclusion

Nous avons décrit dans ce chapitre la procédure ACD de Casares et al. [14]. Cette méthode a été implémentée dans deux outils (Spot et Owl) et nous vu qu'en pratique, elle est plus rapide que `to_parity` malgré l'optimalité de son résultat.

7. Réduction de machine de Mealy incomplètement spécifiée

Le but de ce chapitre est de présenter des méthodes pour réduire la taille d'une machine de Mealy. Cette réduction nous est utile puisqu'un de nos objectifs est de construire une solution du problème de la synthèse LTL qui est la plus simple possible, c'est-à-dire celle avec le moins d'états lorsqu'elle est vue comme une machine de Mealy. Les méthodes qui vont être décrites ici ont été présentées dans l'article « Effective Reductions of Mealy Machines » [67] présenté à FORTE'22. Il s'agit d'un travail conjoint avec Philipp Schlehuber-Caissier. Ce dernier s'est occupé de la procédure de minimisation alors que mon travail a porté sur les méthodes basées sur la bisimulation.

7.1. Contexte

Lorsque `ltlsynt` a trouvé une stratégie (si elle existe) pour le joueur 1, il en extrait un automate découpé possédant plusieurs propriétés. Il s'agit d'abord d'un ω-automate où toutes les exécutions sont acceptantes (il n'y a donc plus de coloration). Une deuxième propriété est ce que nous allons décrire comme une complétion sur les entrées indiquant que pour tout état du joueur 0 et toute combinaison des variables d'entrée, il existe une arête portant cette combinaison partant de cet état. De plus pour tout état du joueur 1, il existe une unique arête qui en part.

Cette idée de complétion sur les entrées et de l'acceptation de toutes les exécutions peut nous rapprocher des machines de Mealy. Il s'agit intuitivement d'une machine qui à toute suite d'entrées associe une suite de sorties.

Pour en donner un premier aperçu, appuyons-nous sur la stratégie de la figure 7.1a. Dans cet automate, depuis l'état 0, on peut aller vers l'état 1 avec $\bar{i}$ alors que depuis l'état 1, l'unique destination possible est l'état 0 et est associée à o. On va donc dire que pour aller de 0 à 0 en ayant lu i, on a produit o, ce qui correspond à l'arête portant l'étiquette $\bar{i}/o$. L'arête portant $i/\bar{o}$ est construire de la même manière.

Remarque 57. *La machine de Mealy est donc une représentation semblable à ce qui est utilisé avant le découpage (voir page 47). Cependant, il y a une distinction sur chaque arête des variables d'entrées et de sorties.*

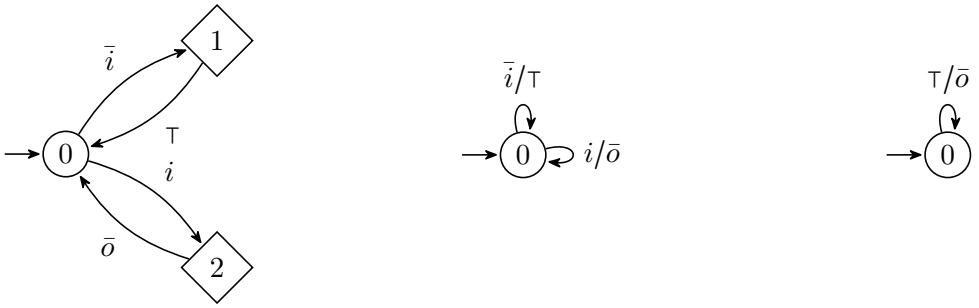
La minimisation de telles machines est analogue à celle des automates finis et est donc relativement simple [36].

Attardons-nous maintenant sur la stratégie de la figure 7.2a. Appliquer la même idée fait que nous obtenons la machine de la figure 7.2b. Celle-ci indique que lorsque l'on voit $\bar{i}$, on peut produire o ou $\bar{o}$. Par définition une machine de Mealy associe à une suite

(a) Exemple de stratégie

(b) Machine de Mealy associée à la stratégie de la figure 7.1a

FIGURE 7.1. – Exemple de transformation d'une stratégie en machine de Mealy



(a) Exemple de stratégie

(b) Machine de Mealy associée à la stratégie de la figure 7.1a

(c) Machine de Mealy minimale compatible avec celle de la figure 7.2b

FIGURE 7.2. – Exemple de transformation d'une stratégie en machine de Mealy incomplètement spécifiée

d'assignations des variables d'entrées une suite d'assignations des variables de sorties [55]. Ici la machine produit une suite d'ensembles d'assignations possibles des variables de sortie. Ce n'est donc pas une machine de Mealy.

Il nous faut alors introduire une machine plus générique que nous désignerons par machine de Mealy incomplètement spécifiée (IGMM) pour permettre ce choix de sortie.

L'intuition derrière la minimisation est alors différente. L'idée est qu'il est possible de restreindre les choix de sortie de ces machines pour pouvoir obtenir une machine minimale. Par exemple, dans la figure 7.2c, nous avons restreint la sortie de l'arête portant $\bar{i}/\top$ à $\bar{o}$. Ainsi, toutes les arêtes sont associées à la même sortie.

À la différence des machines de Mealy classiques, la minimisation d'IGMM telle que nous venons de la décrire est un problème NP-complet [59].

Même si l'obtention d'une stratégie plus simple (en nombre d'états) peut être intéressante, nous avons évoqué l'intérêt important pour la production de circuit AIG. Comme nous l'avons indiqué à la page 54, ce circuit est créé à partir de cette IGMM. Pour com-

prendre l'impact de cette minimisation sur la taille du circuit, nous invitons le lecteur à consulter l'annexe A.

Le but de ce chapitre est tant d'améliorer une méthode existante que de considérer un problème plus simple qu'est la réduction, c'est-à-dire essayer de réduire la taille d'une IGMM sans forcément obtenir un résultat optimal.

7.2. Définitions

Introduisons de manière plus formelle les machines de Mealy incomplètement spécifiées ainsi que les notions de variation et de spécialisation entre les états de celles-ci. Ce sera également l'occasion d'introduire les problèmes de réduction et de minimisation d'IGMM.

Définition 69 (Machine de Mealy incomplètement spécifiée)**.** *Une machine de Mealy incomplètement spécifiée (IGMM) est un tuple* $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ *où*

- I *est un ensemble de variables d'entrées ;*
- O *est un ensemble de variables de sorties ;*
- Q *est un ensemble fini d'états ;*
- q_{init} *est un état initial ;*
- $\delta : (Q, \mathbb{B}^I) \to Q$ *est une fonction de transition partielle ;*
- $\lambda : (Q, \mathbb{B}^I) \to (2^{\mathbb{B}^O} \setminus \varnothing)$ *est une fonction de sortie telle que* $\lambda(q, i) = \top$ *lorsque* $\delta(q, i)$ *n'est pas définie. Si* δ *est une fonction complète, on dit que* $\mathcal{M}$ *est* complète sur les entrées.

Remarque 58. *Certaines définitions d'IGMM autorisent plus de liberté. L'état initial peut par exemple ne pas être spécifié [1].*

Notation 9 (Transition d'une IGMM)**.** *Soit* $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ *une IGMM. Pour tout* $u \in \mathbb{B}^I$ *et* $q \in Q$*, si* $\delta(q, u)$ *est définie, on écrit* $q \xrightarrow{u/v} \delta(q, u)$ *pour tout* $v \in \lambda(q, u)$*.*

Décrivons maintenant la sémantique associée à une telle machine :

Définition 70 (Sémantique d'une IGMM)**.** *Soit* $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ *une IGMM. Étant données deux séquences infinies de valuations* $\iota = i_0 \cdot i_1 \cdots \in (\mathbb{B}^I)^\omega$ *et* $o = o_0 \cdot o_1 \cdots \in (\mathbb{B}^O)^\omega$*, on note* $(\iota, o) \vDash \mathcal{M}_q$ *si et seulement si :*

- *Soit il existe une séquence infinie d'états* $(q_j)_{j \geq 0} \in Q^\omega$ *telle que* $q = q_0$ *et* $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} \dots$ *;*
- *ou il existe une suite finie d'états* $(q_j)_{0 \leq j \leq k} \in Q^{k+1}$ *telle que* $q = q_0$*,* $\delta(q_k, i_k)$ *n'est pas défini et* $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} \dots q_k$*.*

On dit qu'à partir de l'état q*, étant donnée l'entrée* ι*,* $\mathcal{M}$ *produit* o*.*

Remarque 59. *La définition de la sémantique d'une IGMM implique que si $\delta(q_k, i_k)$ n'est pas défini, alors la machine est autorisée à produire une sortie arbitraire à partir de cet état.*

De plus étant donnée une entrée ι il peut exister plusieurs sorties o telles que $(\iota, o) \vDash \mathcal{M}_q$.

Introduisons maintenant deux relations entre les états d'une IGMM établissant des liens entre les sorties qui peuvent être produites à partir de ces états.

Définition 71 (Variation et spécialisation)**.** *Soient $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ et $\mathcal{M}' = (I, O, Q', q'_{init}, \delta', \lambda')$ deux IGMM. Étant donnés deux états $q \in Q$ et $q' \in Q'$, on dit que q' est une :*

- variation *de q si $\forall \iota \in (\mathbb{B}^I)^\omega, \{o \mid (\iota, o) \vDash \mathcal{M}'_{q'}\} \cap \{o \mid (\iota, o) \vDash \mathcal{M}_q\} \neq \varnothing$;*
- spécialisation *de q si $\forall \iota \in (\mathbb{B}^O)^\omega \{o \mid (\iota, o) \vDash \mathcal{M}'_{q'}\} \subseteq \{o \mid (\iota, o) \vDash \mathcal{M}_q\}$.*

On dit que $\mathcal{M}'$ est une variation (resp. spécialisation) de $\mathcal{M}$ si q'_{init} est une variation (resp. spécialisation) de q_{init}.

De manière intuitive, tous les couples entrées-sorties acceptés par une spécialisation q' de $\mathcal{M}'$ sont également acceptés par $q \in Q$. Pour que deux états soient la variation l'un de l'autre, il faut que pour toute suite d'entrées ils soient capables de se mettre d'accord sur une sortie commune.

Notation 10. *On écrit $q' \approx q$ (resp. $q' \sqsubseteq q$) si q' est une variation (resp. spécialisation) de q.*

Notons que $\approx$ est une relation symétrique non-transitive alors que $\sqsubseteq$ est transitive ($\sqsubseteq$ est un préordre).

Comme indiqué plus tôt, nous cherchons à réduire la taille d'une IGMM. Deux types de réductions seront abordés :

La *réduction* d'une IGMM $\mathcal{M}$ consistant à trouver une spécialisation de $\mathcal{M}$ ayant au plus le même nombre d'états mais de préférence moins.

La *minimisation* d'une IGMM $\mathcal{M}$ consistant à trouver une spécialisation de $\mathcal{M}$ ayant le plus petit nombre d'états possible.

Avant de poursuivre nous allons évoquer les différentes représentations de machine de Mealy qui seront traitées. La définition d'IGMM indique que la fonction de sortie produit un élément de $2^{\mathbb{B}^O} \setminus \varnothing$. Nous avons vu page 19 qu'un tel élément peut être vu comme une disjonction de cubes. La fonction de sortie d'une IGMM pourra donc être de la forme $\lambda : Q \times \mathbb{B}^I \to 2^{\mathbb{K}^O}$.

Par exemple, posons $I = \{a\}$ et $O = \{x, y, z\}$. L'ensemble des valuations $v = \{\bar{x}yz, \bar{x}y\bar{z}, x\bar{y}\bar{z}, x\bar{y}z\} \in 2^{\mathbb{B}^O}$ est équivalent à l'ensemble de cubes $v_c = \{\bar{x}y, x\bar{y}\} \in 2^{\mathbb{K}^O}$.

Une machine de Mealy associe usuellement une valuation d'entrée à une unique valuation de sortie. Il s'agit donc d'une fonction de la forme $\lambda : Q \times \mathbb{B}^I \to \mathbb{B}^O$. L'outil MeMin [1] utilise une légère modification de cette définition en permettant de définir une sortie à l'aide d'un cube. La fonction de sortie est alors $\lambda : Q \times \mathbb{B}^I \to \mathbb{K}^O$. Ainsi,

contrairement à notre modèle, tant la définition usuelle de machine de Mealy que celle de MeMin ne permet de représenter l'ensemble v puisqu'il ne peut pas être représenté par un unique cube ou une valuation. Notre modèle est donc strictement plus expressif.

Une optimisation commune à MeMin et notre implémentation concerne la possibilité de fusionner des arêtes : si deux arêtes partagent la même source, la même destination et la même valuation de sortie, alors elles peuvent être fusionnées. Cela ne change pas l'expressivité des modèles mais permet une représentation plus succincte des machines, ce qui améliore l'efficacité des algorithmes.

7.3. État de l'art

Divers outils tels que BICA [58], STAMINA [68] ou COSME [2] existent pour minimiser une IGMM. Cependant, Abel et Reineke ont montré [1] que leur outil, MeMin est bien plus efficace que ces trois méthodes et ce sera donc le seul outil auquel nous nous comparerons.

Nous allons ici seulement rappeler une différence entre leur modèle et le nôtre qui aura un impact sur les résultats.

Il s'agit du choix de représenter dans MeMin une sortie à l'aide d'un cube. Cela signifie que contrairement à notre modèle, MeMin n'est pas capable de gérer un ensemble quelconque d'assignations des variables de sorties.

Nous verrons que ce choix permet à MeMin de se passer d'une condition de non-vacuité de sortie imposée par l'utilisation de notre modèle plus expressif.

Le lecteur souhaitant connaître l'idée de base de cet outil est invité à lire l'annexe C.

7.4. Minimisation d'IGMM

7.4.1. Principe

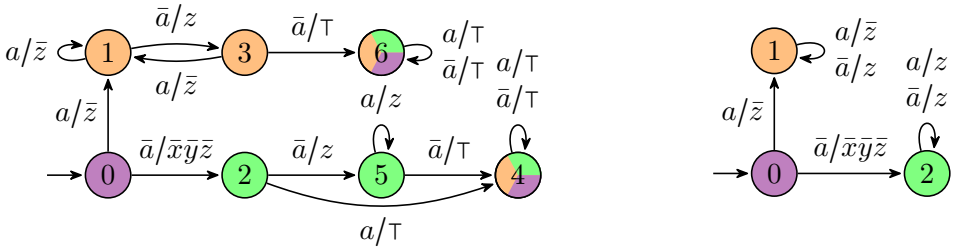
Dans cette section nous allons introduire la minimisation des IGMM. Cette méthode s'inspire de celle utilisée dans l'outil MeMin de Abel et Reineke. Commençons tout d'abord par introduire diverses notions.

Définition 72 (Classe de variation)**.** *Étant donnée une IGMM $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$, une classe de variation $C \subseteq Q$ est un ensemble d'états qui sont des variations les uns des autres, c'est-à-dire si $\forall q, q' \in C, q \approx q'$. Pour toute entrée $i \in \mathbb{B}^I$, on définit*

- *la fonction successeur* $\mathrm{Succ}(C, i) = \bigcup_{q \in C} \{\delta(q, i) \mid \delta(q, i) \text{ est défini}\}$;
- *la fonction sortie* $\mathrm{Out}(C, i) = \bigcap_{q \in C} \lambda(q, i)$.

Définition 73 (Condition de couverture)**.** *On dit qu'un ensemble de classe de variation S couvre une IGMM M si tout état de $\mathcal{M}$ apparaît dans au moins une de ces classes.*

Définition 74 (Condition de clôture)**.** *On dit qu'un ensemble de classes S est clos si pour tout $C_j \in S$ et pour toute entrée $i \in \mathbb{B}^I$ il existe un $C_k \in S$ tel que $\mathrm{Succ}(C_j, i) \subseteq C_k$.*



(a) IGMM de départ

(b) Spécialisation minimale de l'IGMM de la figure 7.3a

FIGURE 7.3. – Exemple de minimisation d'IGMM

Définition 75 (Condition de non-vacuite de sortie)**.** *On dit qu'une classe de variation C a une sortie non-vide si $\mathrm{Out}(C, i) \neq \varnothing$ pour toute entrée $i \in \mathbb{B}^I$.*

Remarque 60. *La condition de non-vacuité de sortie est strictement plus forte que la condition d'avoir tous les éléments d'une classe qui sont des variations les uns des autres. Cette distinction aura son importante puisqu'elle nous permettra de réduire la durée d'exécution de cette méthode.*

Nous allons maintenant adapter la construction de ABEL et REINEKE pour le cas des IGMM en y apportant la condition de non-vacuité de sortie.

Théorème 10. *Soit $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ une IGMM et $S = \{C_0, \ldots, C_{n-1}\}$ un ensemble minimal (en taille) des classes de variation tel que*

- *S est clos ;*
- *S couvre $\mathcal{M}$;*
- *chacune des classe C_j a une sortie non-vide.*

Soit alors $\mathcal{M}' = (I, O, S, q'_{init}, \delta', \lambda')$ une IGMM telle que :

- *$q'_{init} = C$ pour un certain $C \in S$ tel que $q_{init} \in C$;*
- *$\delta'(C_j, i) = \begin{cases} C_k \text{ pour un certain } k \text{ tel que } \mathrm{Succ}(C_j, i) \subseteq C_k & \text{si } \mathrm{Succ}(C_j, i) \neq \varnothing \\ \text{non défini} & \text{sinon} \end{cases}$;*
- *$\lambda'(C_j, i) = \begin{cases} \mathrm{Out}(C_j, i) & \text{si } \mathrm{Succ}(C_j, i) \neq \varnothing \\ \top & \text{sinon} \end{cases}$.*

Alors $\mathcal{M}'$ est une spécialisation de taille minimale (en termes de nombre d'états) de $\mathcal{M}$.

Illustrons maintenant cette construction sur un exemple.

Exemple 21 (Construction d'une IGMM minimale à l'aide d'un SAT-solver, adapté de [1])**.** *Minimisons l'IGMM de la figure 7.3a. Dans cette machine, l'ensemble des propositions d'entrée est $I = \{a\}$ (et donc $\mathbb{B}^I = \{a, \bar{a}\}$) et l'ensemble des propositions de sortie est $O = \{x, y, z\}$. Afin de simplifier les notations, les éléments de $2^{\mathbb{B}^O}$ sont représentés*

comme des fonctions Booléennes (des cubes dans cet exemple) plutôt que comme des ensembles.

Les états ont été colorés de manière à indiquer leur possible appartenance à une des trois classes de variation.

Le SAT-solveur doit alors associer à chaque état au moins une d'entre elles de manière à satisfaire la condition de couverture tout en respectant les contraintes de clôture et de non-vacuité de sortie.

Un des choix possibles est

— $C_0 = \{0\}$;

— $C_1 = \{1, 3, 6\}$;

— $C_2 = \{2, 4, 5\}$.

On peut remarquer qu'avec une telle solution, la classe violette C_0 respecte bien la condition de clôture puisqu'elle ne contient qu'un état. De plus toutes les transitions des états de la classe orange C_1 vont dans les états de C_1 donc la condition y est aussi vérifiée. La même chose peut être vérifiée pour la classe verte C_2.

Enfin, vérifions la condition de non-vacuité de sortie. Encore une fois, c'est vérifié pour la classe C_0. Pour les classes orange et verte, il faut calculer leur sortie respective. On obtient $\mathrm{Out}(C_1, a) = \{\bar{z}\}$, $\mathrm{Out}(C_1, \bar{a}) = \{z\}$, $\mathrm{Out}(C_2, a) = \{\bar{z}\}$, $\mathrm{Out}(C_2, \bar{a}) = \{z\}$. *Puisqu'aucun de ces ensembles n'est vide, la condition de non-vacuité de sortie est satisfaite.*

L'automate construit à partir de cette solution est décrit dans la figure 7.3b.

Remarque 61. *Dans cet automate, la réduction ne s'appuie que sur le choix des sorties. Seul ce type de cas nous concerne puisque dans le cadre de la minimisation des stratégies, nos IGMM sont complètes sur les entrées.*

7.4.2. Encodage SAT proposé

Afin de construire une spécialisation minimale d'une IGMM $\mathcal{M}$, nous utiliserons l'approche itérative suivante dans laquelle n vaut 1 au départ :

— Supposer qu'il y a n classes et donc que la machine minimale a n états ;

— Créer des clauses imposant que les conditions de couverture, de clôture et de non-vacuité de sortie soient respectées ;

— Vérifier si le problème SAT obtenu est satisfaisable ;

— Si tel est le cas, appliquer la construction du théorème 10 ;

— Sinon, incrémenter n et appliquer de nouveau le processus avec cette valeur.

Remarque 62. *Si $n = |Q|$, il n'y a pas besoin de chercher une solution puisque cela signifie que la machine est déjà minimale.*

Il nous faut alors indiquer comment sont construites les conditions associées à la deuxième étape.

Encodage des conditions de couverture et de clôture

Afin d'assurer qu'un ensemble de classes $C = \{C_0, \ldots, C_{n-1}\}$ vérifie les conditions de couverture et de clôture mais aussi d'imposer que chaque classe C_j est une classe de variation, nous allons introduire deux types de littéraux :

— $s_{q,j}$ qui doit être vrai si et seulement si l'état q appartient à la classe C_j ;

— $z_{i,k,j}$ qui doit être vrai si $\text{Succ}(C_k, i) \subseteq C_j$ pour $i \in \mathbb{B}^I$.

La condition de couverture est alors encodée par l'équation 7.1 puisqu'elle garantit que tout état appartient à au moins une classe.

$$\bigwedge_{q \in Q} \bigvee_{0 \leq j < n} s_{q,j} \qquad (7.1)$$

$$\bigwedge_{0 \leq j < n} \bigwedge_{\substack{q,q' \in Q \\ q \not\approx q'}} \overline{s_{q,j}} \vee \overline{s_{q',j}} \qquad (7.2)$$

L'équation 7.2 impose à chaque classe d'être une classe de variation : deux états q et q' qui ne sont pas la variation l'un de l'autre ne peuvent pas correspondre à la même classe.

La condition de clôture doit vérifier que pour chaque classe C_k et tout symbole d'entrée $i \in \mathbb{B}^I$, il existe au moins une classe C_j contenant tous les successeurs : $\forall k, \forall i, \exists j, \text{Succ}(C_k, i) \subseteq C_j$. Cela est exprimé par les contraintes 7.3 et 7.4.

$$\bigwedge_{0 \leq k < n} \bigwedge_{i \in \mathbb{B}^I} \bigvee_{0 \leq j < n} z_{i,k,j} \qquad (7.3)$$

$$\bigwedge_{0 \leq j,k < n} \bigwedge_{\substack{q,q' \in Q, i \in \mathbb{B}^I \\ q' = \delta(q,i)}} (z_{i,k,j} \wedge s_{q,k}) \rightarrow s_{q',j} \qquad (7.4)$$

Plus précisément, la contrainte 7.3 assure qu'au moins une classe C_j contient $\text{Succ}(C_k, i)$ alors que la contrainte 7.4 impose que cette association respecte les transitions de $\mathcal{M}$.

Encodage de la condition de non-vacuité de sortie

Comme nous l'avions indiqué dans la remarque 60, vérifier que chaque classe de S est une classe de variation est nécessaire mais pas suffisant pour satisfaire la condition de non-vacuité de sortie. On veut en réalité imposer que chaque entrée i, tous les états dans une classe donnée peuvent se mettre d'accord sur au moins une assignation des variables de sortie.

Cependant il est possible d'avoir au moins trois états (par exemple $\textcircled{0}\!\circlearrowright a/\{xy, x\bar{y}\}$, $\textcircled{1}\!\circlearrowright a/\{\bar{x}y, x\bar{y}\}$, et $\textcircled{2}\!\circlearrowright a/\{xy, \bar{x}y\}$) qui sont tous les variations les uns des autres mais ne peuvent s'accorder sur un comportement commun.

Cette situation ne peut pas se produire avec MeMin puisque leur modèle utilise des cubes pour représenter les sorties plutôt que des ensembles quelconques de valuations des variables de sortie comme dans le nôtre. Une propriété utile des cubes est que si les intersections des cubes sont deux à deux non-vides, alors l'intersection de ces cubes l'est également.

Puisque les cubes ne sont pas assez expressifs pour notre modèle, il nous faut alors généraliser les sorties comme indiqué page 130. Un ensemble arbitraire de valuations est alors représenté par un ensemble de cubes dont la disjonction correspond à l'ensemble de départ.

Pour obtenir cette représentation, nous nous appuyons sur l'algorithme de Minato [56] pour transformer une fonction Booléenne représentée par un BDD en une somme de cubes non-redondants et notons l'ensemble des cubes obtenus par $\mathrm{SC}(\lambda(q,i))$.

Notre approche pour assurer qu'il existe une assignation commune est de rechercher des cubes disjoints et les exclure des sorties possibles en les désactivant si nécessaire. Un cube actif est un ensemble dans lequel on va chercher une valuation des variables de sortie sur laquelle la classe peut s'accorder. Pour l'exprimer, introduisons deux types de littéraux :

— $a_{c,q,i}$ qui doit être vrai si et seulement si l'instance particulière du cube $\mathrm{SC}(\lambda(q,i))$ utilisée dans la sortie de l'état q en lisant i est active ;

— $sc_{q,q'}$ doit être vrai si et seulement s'il existe une classe $C_j \in S$ telle que $q \in C_j$ et $q' \in C_j$.

La désactivation sélective d'un cube peut être exprimée par les équations suivantes :

$$\bigwedge_{\substack{q,q' \in Q \\ 0 \leq j < n}} (s_{q,j} \wedge s_{q',j}) \rightarrow sc_{q,q'} \tag{7.5}$$

$$\bigwedge_{\substack{q \in Q,\, i \in \mathbb{B}^I \\ \delta(q,i) \text{ est défini}}} \bigvee_{c \in \mathrm{CS}(\lambda(q,i))} a_{c,q,i} \tag{7.6}$$

$$\bigwedge_{\substack{q,q' \in Q,\, i \in \mathbb{B}^I \\ \delta(q,i) \text{ est défini} \\ \delta(q',i) \text{ est défini}}} \bigwedge_{\substack{c \in \mathrm{CS}(\lambda(q,i)) \\ c' \in \mathrm{CS}(\lambda(q',i)) \\ c \cap c' = \varnothing}} (a_{c,q,i} \wedge a_{c',q',i}) \rightarrow \overline{sc_{q,q'}}. \tag{7.7}$$

L'équation 7.5 impose que $sc_{q,q'}$ est vrai s'il existe une classe contenant à la fois q et q'. La contrainte 7.6 garantit qu'au moins un des cubes est dans la sortie $\lambda(q,i)$ est actif, ce qui impose à la sortie d'être non-vide. La contrainte 7.7 exprime la désactivation sélective et a seulement besoin d'être ajoutée pour des $q, q' \in Q$ et $i \in \mathbb{B}^I$ tel que $\delta(q,i)$ et $\delta(q',i)$ sont définis. Cette formule assure que s'il existe une classe à laquelle q et q' appartiennent (c'est-à-dire $sc_{q,q'}$ est vrai) mais aussi qu'il existe des cubes disjoints dans la partition de leur sortie respective, alors on désactive au moins l'un d'entre eux : seuls les cubes qui s'intersectent peuvent être activés. Ainsi cette contrainte garantit la condition de non-vacuité de sortie.

Puisqu'encoder un ensemble d'entrées requiert un nombre de cubes exponentiel en le nombre de variables de O, l'encodage précédent utilise

$$\begin{aligned} O(&\underbrace{|Q|}_{7.1} + \underbrace{n \cdot |Q|^2}_{7.2} + \underbrace{|Q|^2 \cdot n}_{7.5} + \underbrace{|Q|^2 \cdot 2^{|I|} \cdot 2^{2|O|}}_{7.7} + \underbrace{n \cdot 2^{|I|}}_{7.3} + \underbrace{n^2 \cdot |\delta|}_{7.4} + \underbrace{|\delta|}_{7.6}) \\ =&O(|Q|^2 \cdot (n + 2^{|I|+2|O|}) \qquad\qquad + n \cdot 2^{|I|} + |\delta| \cdot n^2) \end{aligned}$$

clauses et

$$\begin{aligned} =&O(\underbrace{|Q| \cdot 2^{|I|+|O|}}_{7.6} + \underbrace{|Q| \cdot n}_{7.1} + \underbrace{|Q|^2}_{7.5} + \underbrace{n^2 \cdot 2^{|I|}}_{7.3} + \underbrace{0}_{7.2} + \underbrace{0}_{7.4} + \underbrace{0}_{7.7}) \\ =&O(|Q| \cdot (2^{|I|+|O|} + n + |Q|) \qquad + n^2 \cdot 2^{|I|}) \end{aligned}$$

variables.

Nous utilisons des optimisations supplémentaires pour limiter le nombre de clauses. En particulier, l'approche CEGAR de la section 7.4.3 cherche à éviter l'introduction des contraintes 7.5 à 7.7.

7.4.3. Amélioration des premières optimisations

Construire itérativement le problème SAT à partir de $n = 1$ serait très inefficace. Pour avoir une borne inférieure plus réaliste, on peut s'appuyer sur le fait que si deux états ne sont pas la variation l'un de l'autre, alors ils ne peuvent pas être dans la même classe. Ainsi s'il est possible de trouver k états qui ne sont pas la variation l'un de l'autre, alors on peut en déduire que la solution optimale doit contenir au moins k états. Cette idée fut d'abord introduite par Abel et Reineke [1] cependant nous allons nous appuyer sur une étude plus poussée des contraintes pour réduire le nombre de contraintes et de littéraux.

La condition de non-vacuité de sortie implique la création de beaucoup de littéraux et clauses et nécessite une étape coûteuse de décomposition des ensembles de sorties retournés par la fonction de sortie ($\lambda : Q \times B^I \to 2^{\mathbb{B}^O} \setminus \{\varnothing\}$) en un ensemble de cubes ($\lambda : Q \times B^I \to 2^{\mathbb{K}^O} \setminus \{\varnothing\}$). Pour générer les clauses 7.5 à 7.7, nous allons éviter d'ajouter de telles clauses avec une approche dirigée par les contre-exemples (à la CEGAR) décrite plus loin. En effet, la violation de ces conditions peut être facilement détectée avant même la construction de la machine. Si elle est détectée, un petit ensemble de ces contraintes est ajouté au problème SAT, excluant ainsi cette violation. Dans de nombreux cas cette optimisation réduit grandement le nombre de littéraux et de contraintes nécessaires

À partir de maintenant, considérons une IGMM avec N états $Q = \{q_0, \ldots, q_{N-1}\}$.

Matrice de variation

La première étape est de déterminer quels états ne sont pas des variations les uns des autres de manière à extraire une solution partielle et appliquer des simplifications sur les contraintes. Il s'agit de calculer une matrice de variation de taille $N \times N$ que nous désignerons par mat telle que $\text{mat}[k][\ell] = 1$ si et seulement si $q_k \not\approx q_\ell$ de la manière suivante :

1. Initialiser toutes les entrées de mat à 0 ;
2. Itérer sur toutes les paires (k, ℓ) avec $0 \leq k < \ell < N$. Si $\text{mat}[k][\ell]$ vaut 0, alors vérifier s'il existe $i \in \mathbb{B}^I$ tel que $\lambda(q_k, i) \cap \lambda(q_\ell, i) = \varnothing$. S'il existe, $\text{mat}[k][\ell] \leftarrow 1$;
3. Pour toutes les paires (k, ℓ) telles que la valeur $\text{mat}[k][\ell]$ est passée de 0 à 1, on cherche les prédécesseurs de q_k et q_ℓ qui peuvent atteindre ces états en lisant la même entrée et on les marque comme incompatibles. Formellement, pour toute paire (m, n) avec $m < n$ telle que $\exists i \in \mathbb{B}^I$ telle que $\delta(q_m, i) = q_k$ et $\delta(q_n, i) = q_k$, on assigne 1 à $\text{mat}[m][n]$. On répète ces changements au prédécesseur de (m, n) jusqu'à atteindre un point fixe.

Puisque la relation de variation est symétrique et réflexive, on ne calcule que les éléments au- dessus de la diagonale de la matrice.

L'intuition derrière cet algorithme est que deux états q et q' ne sont pas la variation l'un de l'autre si :

— il existe un symbole d'entrée pour lequel les ensembles de sortie sont disjoints ;

— il existe une paire d'états qui ne sont pas la variation l'un de l'autre et qui peuvent être atteint depuis q et q' avec la même séquence d'entrées.

La complexité de cet algorithme est $O(|Q|^2 \cdot 2^{|I|})$ si on suppose que vérifier si les ensembles de sorties sont disjoints peut être fait en temps constant. Cette hypothèse est incorrecte en général : tester si des cubes sont disjoints a une complexité linéaire en le nombre de propositions d'entrée. D'un autre côté, tester si deux cubes sont disjoints pour une IGMM utilisant des ensembles quelconques de valuations a une complexité exponentielle en le nombre de propositions d'entrée. La complexité accrue est cependant contrebalancée par le côté succinct des ensembles arbitraires utilisés.

Par exemple, étant donné $2m$ propositions de sortie $o_0, \ldots, o_{2m-1}$, considérons l'ensemble des valuations exprimée comme une disjonction de cubes $\bigvee_{0 \leq k < m} o_{2k}\, \overline{o_{2k+1}} \vee \overline{o_{2k}}\, o_{2k+1}$. Un nombre exponentiel de cubes disjoints est nécessaire pour représenter cet ensemble. Ainsi une machine de Mealy non-déterministe étiquetée par des cubes de sorties va impliquer un nombre exponentiel de calculs faits en temps linéaire, alors qu'une IGMM ne va faire qu'un test en temps exponentiel.

Calcul de la solution partielle

La solution partielle correspond à un ensemble d'états tels qu'aucun d'entre eux est la variation d'un autre de cet ensemble. Ainsi aucun d'entre eux ne peut correspondre à la même classe de variation. La taille de cet ensemble est donc une borne inférieure du nombre d'états de la machine minimale.

Trouver la plus grande solution partielle est un problème NP-difficile. On utilise donc une heuristique gloutonne décrite par Abel et Reineke [1]. Pour chaque état q de M, on compte le nombre d'états q' tels que q' n'est pas une variation de q. Désignons ce nombre par nvc_q. On ajoute alors successivement à la solution partielle les états qui ont le plus grand nvc_q mais ne sont la variation d'aucun autre état déjà inséré.

Approche CEGAR pour assurer la condition de non-vacuité de sortie

En supposant qu'une solution satisfaisant la condition de couverture et de clôture a déjà été trouvée, on doit alors vérifier si cette solution vérifie la condition de non-vacuité de sortie. Si c'est le cas, on peut alors construire puis retourner une machine minimale.

Si la condition n'est pas satisfaite, on doit chercher une ou plusieurs combinaisons de classes C_k et symboles d'entrée i tels que $\mathrm{Succ}(C_k, i) = \varnothing$. Pour chaque état de C_k et le symbole d'entrée i, on ajoute les contraintes décrites dans la section 7.4.2. On vérifie ensuite que le problème est toujours satisfaisable. Si ce n'est pas le cas, il faut alors augmenter le nombre de classes pour trouver une solution valide. Sinon soit la solution

Algorithme 6 : Minimisation SAT

Entrée : une machine $M = (I, O, Q, q_{init}, \delta, \lambda)$
Sortie : une spécialisation minimale M' de M

```
   /* Calcul de la matrice de variation                          */
 1 bool[][] mat ← isNotVariationOf(M)
   /* Recherche d'une solution partielle P                       */
 2 P ← extractPartialSol(mat)
 3 clauses ← {}
   /* Utilisation de la borne inférieure tirée de P              */
 4 pour n ← |P| à |Q| − 1 faire
 5     addCoverCondition(clauses, M, P, mat, n)
 6     addClosureCondition(clauses, M, P, mat, n)
       /* Résolution des conditions de couverture et de clôture  */
 7     (sat, solution) ← satSolver(clauses)
 8     tant que sat faire
 9         si verifyNonEmpty(M, solution) alors
10             retourner buildMachine(M, solution)
           /* Ajout des contraintes de non-vacuité pertinentes   */
11         addNonemptinessCondition(clauses, M, solution)
12         (sat, solution) ← satSolver(clauses)
   /* S'il n'y a pas de solution, retourner M                    */
13 retourner copyMachine(M)
```

respecte la contrainte 7.7 et on peut retourner une machine minimale, soit elle ne la respecte pas et le processus d'ajout de contraintes est répété.

Dans les deux cas, ce schéma CEGAR (*counter-example guided abstraction refinement*) assure la terminaison puisque le problème est soit prouvé insatisfaisable ou résolu par une exclusion itérative des violations de la condition de non-vacuité de sortie.

7.4.4. Algorithme général

Les optimisations décrites précédemment conduisent à l'algorithme 6.

Optimisations supplémentaires et comparaison à MeMin

L'algorithme proposé s'appuie sur l'approche générale décrite par Abel et al. [1], tout comme l'encodage SAT pour les conditions de couverture et clôture. On recherche une solution partielle en utilisant des heuristiques similaires et adaptons quelques optimisations présente dans leur code source qui ne sont pas détaillées dans leur papier.

La plus grande différence se situe dans la plus grande expressivité des entrées et des symboles de sortie, ce qui cause des changements significatifs. En particulier, nous avons ajouté la condition de non-vacuité de sortie pour garantir la correction ainsi qu'une implémentation basée sur l'approche CEGAR pour maintenir de bonnes performances.

Les autres améliorations viennent principalement d'un meilleur usage de la solution partielle.

Par exemple, chaque état q de la solution partielle est associé à sa propre classe C_j. Puisque le littéral $s_{q,j}$ correspondant est trivialement vrai, il peut être omis en remplaçant chaque occurrence par *vrai*.

Les états appartenant à la solution partielle ont également d'autres particularités pouvant être exploitées pour réduire le nombre de classes Succ possibles, ce qui réduit encore le nombre de clauses et littéraux nécessaires.

Nous avons donc besoin de moins de littéraux et clauses tout en traitant une construction plus complexe du problème SAT.

7.5. Réduction d'IGMM

Dans cette section nous allons introduire une autre approche plus proche de notre objectif global : la synthèse de manière efficace. En effet, réduire la taille de la stratégie n'est qu'une étape optionnelle visant à obtenir un contrôleur plus simple mais on n'a forcément envie de passer une partie importante du temps à faire ce travail. Nous introduirons donc une autre approche visant à réduire la taille d'une IGMM sans forcément chercher à obtenir une machine optimale.

Cette réduction est basée sur le théorème suivant :

Théorème 11. *Soit $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ une IGMM et $r : Q \to Q$ une application telle que $\forall q \in Q,\ r(q) \sqsubseteq q$. Définissons alors $\mathcal{M}' = (I, O, Q', q'_{init}, \delta', \lambda)$ comme l'IGMM telle que :*

- *$Q' = r(Q)$;*
- *$q'_{init} = r(q_{init})$;*
- *$\delta'(q, i) = r(\delta(q, i))$ pour tout état q et toute entrée i.*

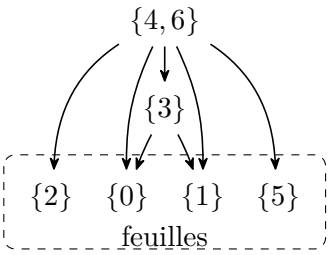
Alors $\mathcal{M}'$ est une spécialisation de $\mathcal{M}$.

De manière intuitive, ce théorème indique que si un état q est remplacé par $r(q)$, alors l'ensemble des sorties qui peuvent être produites pour une entrée donnée est simplement réduit à un sous-ensemble des sorties originales.

Avec cette construction, plus l'image de r est petite, plus la réduction sera significative. Ainsi notre but sera de chercher une telle application r qui associe à chaque état un des éléments minimaux selon le préordre $\sqsubseteq$, également appelé représentant.

Afin de décrire comment sont construites ces applications, introduisons les définitions de graphe de spécialisation ainsi que celle de représentant d'un état.

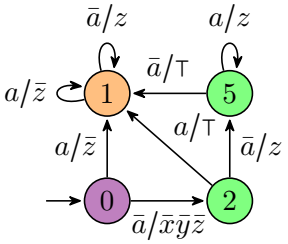
Définition 76 (Graphe de spécialisation d'une IGMM)**.** *Un graphe de spécialisation d'une IGMM $\mathcal{M} = (I, O, Q, q_{init}, \delta, \lambda)$ est le graphe de condensation du graphe acyclique représentant la relation $\sqsubseteq$: les sommets du graphe de spécialisation sont des ensembles formant une partition de Q telle que deux états q et q' correspondent au même sommet si $q \sqsubseteq q'$ et $q' \sqsubseteq q$; il y a une arête $\{q_1, q_2, \ldots\} \to \{q'_1, q'_2, \ldots\}$ si et seulement si $q'_i \sqsubseteq q_j$ pour un (ou de manière équivalente tout) couple i, j.*



(a) Graphe de spécialisation de l'IGMM de la figure 7.3a

q		$r(q)$
0	→	0
1	→	1
2	→	2
3	→	1
4	→	1
5	→	5
6	→	1

(b) Association aux représentant choisie



(c) IGMM obtenue à partir de celle de la figure 7.3a

FIGURE 7.4. – Étapes de réduction de l'IGMM de la figure 7.3a

Par définition ce graphe est acyclique.

Exemple 22 (Graphe de spécialisation d'une IGMM)**.** *Considérons l'IGMM de la figure 7.3a et montrons son lien avec le graphe de spécialisation de la figure 7.4a. On remarque facilement que* $4 \sqsubseteq 6$ *et* $6 \sqsubseteq 4$*. Ces deux états sont donc dans le même nœud du graphe. Ensuite on peut remarquer que l'état 5 est une spécialisation des états 4 et 6. Il y a donc une arête allant vers le nœud portant et qui part du nœud portant 4 et celui portant 6 (qui est le même ici).*

Définition 77 (Représentant d'un état)**.** *Étant donnés deux états* q *et* q' *d'une IGMM* $\mathcal{M}$*,* q' *est un* représentant *de* q *si, dans le graphe de spécialisation de* $\mathcal{M}$*,* q' *correspond à une feuille pouvant être atteinte à partir du sommet contenant* q*. En d'autres mots,* q' *est un représentant de* q *si* $q' \sqsubseteq q$ *et* q' *est minimal selon le préordre* $\sqsubseteq$*.*

Notons qu'un état peut admettre plusieurs représentants et en a forcément au moins un (lui-même).

Exemple 23 (Construction d'une spécialisation à partir d'un graphe de spécialisation)**.** *Appuyons-nous sur le graphe de spécialisation de la figure 7.4a pour construire une spécialisation de l'IGMM de la figure 7.3a.*

La première étape consiste à choisir dans le graphe de spécialisation un représentant pour chaque état. Comme nous l'évoquions page 139, il s'agit de trouver pour chaque état q *le nœud qui le porte puis à choisir une feuille accessible depuis ce nœud. Le*

représentant est alors un état de cette feuille. Par exemple, puisque l'état 1 est dans une feuille accessible depuis le nœud portant 3, alors 1 est le représentant de 3. Le choix des représentants est décrit dans la figure 7.4b.

À partir de cette association, il nous reste alors à construire la spécialisation. Pour cela, il suffit de copier l'IGMM de départ puis de modifier les destinations des arêtes. Ainsi l'arête allant de 3 vers 6 dans l'IGMM de départ va maintenant vers l'état 1. L'IGMM obtenue est alors celle de la figure 7.4c.

Remarque 63. *Il est possible de choisir dans une feuille n'importe quel état. Cependant pour limiter la taille du résultat, il faut pour une feuille donnée toujours choisir le même état.*

Remarque 64. *Deux états dans un même nœud peuvent choisir un représentant différent. Ainsi on peut choisir 0 comme représentant de 4 et 5 comme représentant de 6.*

Nous avons donc vu qu'il est possible de s'appuyer sur la relation de spécialisation pour réduire une IGMM par une procédure que nous désignerons par bisimulation avec spécialisation des sorties. Cependant le théorème 11 indique que toute relation impliquant la spécialisation peut être utilisée pour la réduction. Nous allons donc proposer une autre relation. Il s'agit de la relation de *bisimulation*. L'idée est analogue à la minimisation d'automate fini : si deux états sont indistinguables (peuvent produire le même ensemble de sortie pour toute suite d'entrées), alors ils peuvent être fusionnés. Pour la décrire, appuyons-nous de nouveau sur l'IGMM de la figure 7.3a. Dans cette IGMM, les deux seuls états qui ont le même comportement sont 4 et 6. On va donc choisir 4 comme représentant de 6 et tous les autres états seront leur représentant.

Remarque 65. *Avec une telle relation l'IGMM résultante est équivalente à celle de départ.*

Remarque 66. *Le nom de bisimulation avec spécialisation des sorties vient de l'équivalence de cette procédure à la restriction des sorties sur certaines arêtes avant d'appliquer la réduction basée sur cette bisimulation.*

7.5.1. Implémentation

Décrivons maintenant l'implémentation de la procédure de décision pour la bisimulation avec spécialisation des sorties.

Pour cela mettons en avant que la définition de $\sqsubseteq$ peut être décrite de manière récursive comme une relation de simulation.

Supposons sans perte de généralité que la machine à réduire est complète sur les entrées. On décrit $\sqsubseteq$ comme la relation satisfaisant

$$q' \sqsubseteq q \Rightarrow \forall i \in \mathbb{B}^I, \begin{cases} \lambda(q', i) \subseteq \lambda(q, i) \\ \delta(q', i) \sqsubseteq \delta(q, i) \end{cases}$$

Cette définition nous permet de décider $\sqsubseteq$ à l'aide de techniques utilisées pour calculer les relations de simulation [34, 26]. Notre implémentation s'appuie sur une adaptation

de la technique des signatures décrite par Babiak et al. [4, Sec. 4.2] : à chaque état q est associé une signature $\mathrm{sig}(q)$ qui est une formule Booléenne (représentée par un BDD) encodant les transitions en sortant tel que $\mathrm{sig}(q') \to \mathrm{sig}(q)$ si et seulement si $q' \sqsubseteq q$. À l'aide des signatures il devient alors facile de construire le graphe de spécialisation et d'en déduire des représentants.

Donnons maintenant une définition de la *signature* dans le cadre des IGMM. Celle-ci est calculée itérativement avec :

$$\mathrm{sig}^i(q) = \bigvee_{(q,i,d)\in\delta} \left(\ell \wedge \lambda(q,i) \wedge \mathrm{Implied}^i(d)\right)$$

$$\mathrm{Implied}^i(d) = \bigwedge_{\substack{C_k^i \in P^i \\ \mathrm{sig}^{i-1}(d)\to\mathrm{sig}^{i-1}(C_k^i)}} \hat{C}_k$$

où $P^i = \{C_1^i, \ldots, C_m^i\}$ est une partition de Q et $\hat{C}_k$ est une variable Booléenne indiquant l'appartenance à l'ensemble C_k^i. On impose $\mathrm{Implied}^0 = \hat{C}_1$ pour tout état $q \in Q$ à la première itération. Le calcul termine lorsque $P^k = P^{k+1}$ et $\mathrm{Implied}^k = \mathrm{Implied}^{k+1}$.

7.6. Évaluation expérimentale

La machine utilisée pour les mesures qui seront présentées est composée de

— un processeur Ryzen 4800HS ;

— 16Go de RAM DDR4.

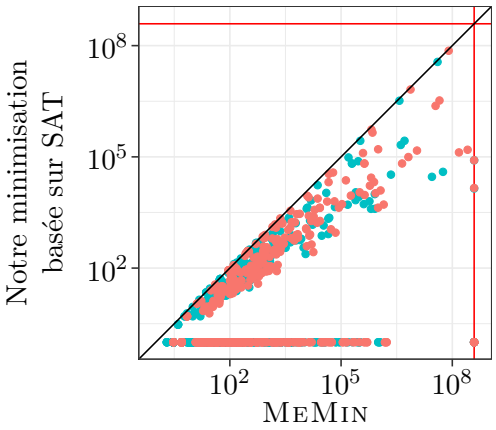
Cette section vise à comparer nos méthodes de réduction et minimisation à MeMin. Ce dernier s'est montré supérieur [1] aux outils existants tels que Stamina [68], Bica [58], et Cosme [2]. C'est pour cela qu'il sera notre unique représentant des outils existants.

Nous nous appuierons sur trois ensembles de machines de Mealy. Les deux premiers étaient déjà utilisés par Abel et Reineke. Il s'agit des ensembles de machines ISM [41] et MCNC [74]. Cependant ces machines sont relativement petites et MeMin peut les minimiser en moins d'une seconde. Nous ajoutons alors des stratégies obtenues avec `ltlsynt` pour des cas de la SYNTCOMP [39].

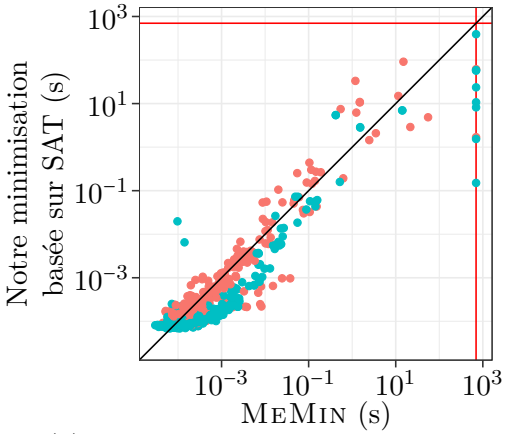
À la différence des machines des deux autres ensembles, les stratégies portent des sorties qui ne peuvent pas être exprimées par un unique cube. Comme nous l'avons indiqué page 130, MeMin n'est pas capable d'exprimer une telle liberté et les machines doivent alors être restreintes pour pouvoir être traitées par cet outil.

L'ensemble des cas liés à la SYNTCOMP sera donc décliné en deux versions. La première, qui sera désignée par full contiendra les machines telles que données par l'outil alors la seconde, qui sera désignée par cube, restreindra les sorties de ces machines à un cube pour pouvoir être traités par MeMin.

Remarque 67. *La version cube des machines est obtenue en restreignant les sorties possibles. Ainsi, MeMin peut produire une machine plus grande que celle obtenue avec notre outil pour la version full.*



(a) Comparaison du nombre de clauses (en rouge) et du nombre de variables (en bleu) des problèmes SAT qui sont créés par MeMin et notre outil



(b) Comparaison de la durée totale traitement de notre minimisation avec MeMin. Les points en bleu sont ceux pour lesquels notre minimisation n'a pas eu besoin de résoudre de problème SAT.

FIGURE 7.5. – Comparaison de la complexité des problèmes SAT de MeMin et notre minimisation ainsi que de la durée totale de traitement. Les lignes rouges indiquent les cas qui n'ont pas terminé.

7.6.1. Différences d'encodage entre MeMin et notre minimisation

Commençons par une comparaison de la taille des problèmes SAT qui doivent être résolus par les deux outils. Pour cela, les deux outils utilisent les versions cube des machines.

Pour cela, appuyons-nous sur la figure 7.5a. Il s'agit d'une comparaison du nombre de clauses (en rouge) et variables (en bleu) du plus grand problème SAT que les deux outils doivent résoudre. Une limite de 1800 secondes a été imposée. Les points sur les lignes rouges indiquent que l'outil n'a pas terminé. Ces cas ne concernent que MeMin et il s'agit de cas où le programme n'a pas réussi à allouer suffisamment de mémoire.

La première zone à mettre en évidence se situe tout en bas. Il s'agit d'un ensemble de cas pour lesquels notre minimisation n'a pas besoin de résoudre de problème SAT. Cela vient du fait que nous arrivons à détecter que la machine est déjà minimale à partir de la solution partielle. MeMin n'en est pas capable, ce qui implique parfois la création de très grandes formules ou même que le programme n'est pas capable de terminer. Cela représente 378 des 644 cas traités.

Pour les cas où les deux outils ont besoin de résoudre un problème SAT, on voit que MeMin a toujours (264 cas) besoin de plus de variables et de clauses que notre minimisation. Cet écart est important puisque MeMin produit des problèmes ayant entre 1,05 et 1667 fois (20,09 en moyenne) plus de clauses.

Maintenant que l'on sait que notre minimisation a besoin de résoudre des problèmes plus petits, voyons si cela a un impact sur la durée de traitement. Pour cela, appuyons-

nous sur la figure 7.5b. Il s'agit d'une comparaison du temps total de traitement pour les deux outils où la coloration met en évidence les cas pour lesquels notre minimisation n'a pas eu besoin de résoudre de problème SAT (en bleu).

Remarque 68. *Ne pas créer un problème SAT n'est pas équivalent à la minimalité de la machine de départ. Pour 7 cas, la machine n'est pas minimale mais il n'y a pas eu besoin de SAT-solver car la solution partielle donnait déjà la machine minimale.*

La première chose qui est visible est que pour une grande partie des cas, les deux outils sont capables de traiter rapidement le problème (620 cas parmi les 644 peuvent être traités par les deux outils en moins d'une seconde). Parmi ces cas, on peut remarquer qu'en général, lorsque notre minimisation n'a pas besoin de résoudre de problème SAT, elle est la plus rapide. Lorsqu'un tel problème doit être résolu, il devient plus difficile de départager les deux outils.

Si on considère les cas que les deux outils sont capables de traiter en plus d'une seconde, on remarque que même lorsque notre minimisation n'a pas besoin de résoudre un problème SAT, cela ne lui confère pas un avantage par rapport à MeMin.

Cependant à droite de la figure il existe un ensemble de cas que notre minimisation traite en plus d'un dixième de seconde alors que MeMin ne termine pas. Il s'agit presque exclusivement de cas où l'étude de la solution partielle évite la construction du problème SAT.

On peut donc conclure que les différentes optimisations apportées à notre procédure par rapport à MeMin (hors généralisation du modèle) ont conduit à un gain de performances. Ce lien étant principalement lié à la détection de cas pour lesquels l'utilisation d'un SAT-solver n'est pas nécessaire.

7.6.2. Impact de la spécialisation des sorties sur la bisimulation

Nous allons maintenant voir l'impact de la spécialisation des sorties sur la taille de la machine résultante. Pour cela nous allons nous appuyer sur la figure 7.6.

Il s'agit d'une comparaison de la distribution des rapports entre la machine obtenue par les réductions basées sur la bisimulation et la solution optimale. Puisque MeMin est exclu, il n'y a que les machines full et nous nous restreignons aux stratégies.

On y voit que la bisimulation seule permet à obtenir une machine minimale pour 40 des 316 cas qui n'étaient pas minimaux. Les machines sont en moyenne 2,71 fois plus grandes que le résultat optimal.

Utiliser la spécialisation des sorties permet d'obtenir 232 résultats optimaux. De plus les machines sont alors en moyenne 1,5 fois plus grandes que l'optimale.

Même s'il est parfois possible d'obtenir une machine minimale par une simple bisimulation, on voit donc que l'utilisation de la spécialisation des sorties permet l'obtention de beaucoup plus de machines minimales.

Sur les cas où la bisimulation produit une machine beaucoup plus grande que la machine optimale, on voit cependant que l'apport de la spécialisation des sorties est moins important même s'il reste perceptible.

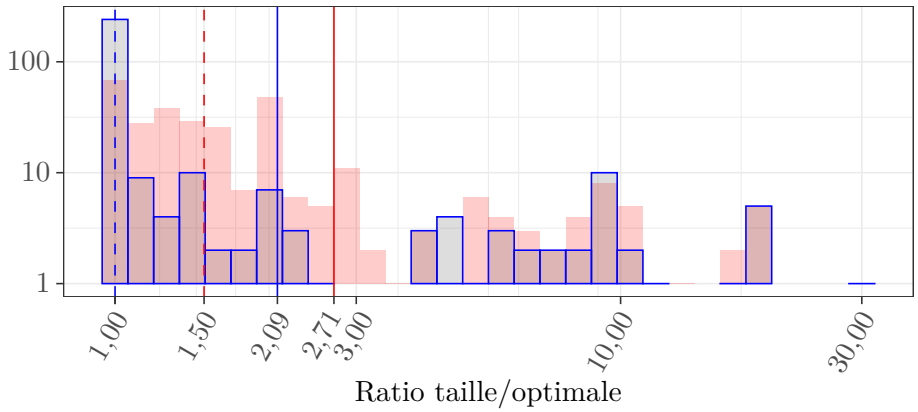


FIGURE 7.6. – Histogramme du rapport entre la taille après la réduction et la taille d'une machine optimale pour les cas où la machine de départ n'est pas déjà minimale. La bisimulation est en rouge alors que la bisimulation avec spécialisation des sorties est en bleu. Les lignes continues représentent les moyennes pour chaque algorithme alors que les lignes en pointillés représentent les médianes.

7.6.3. Comparaison des différentes méthodes entre elles

Nous allons ici nous placer dans le cadre qui intéresse, la synthèse LTL. Nous avons déjà indiqué que le modèle utilisé par MeMin n'est pas capable d'exprimer toute la liberté contenue dans une stratégie. Pour montrer ce qu'apporte notre modèle plus expressif, seules les machines données à MeMin sont modifiées pour qu'il puisse les traiter. Nous nous restreignons à l'ensemble des machines qui sont des stratégies obtenues à partir des formules de la SYNTCOMP et que MeMin a réussi à traiter.

Nous allons nous appuyer sur le tableau 7.1 qui compare les méthodes entres elles mais aussi par rapport à la machine optimale. Pour montrer l'impact de la plus grande expressivité de notre modèle, nos méthodes utiliseront les versions full des machines de Mealy alors que MeMin utilisera la version cube.

Dans ce cadre, la notion de machine minimale est définie comme étant le résultat de notre minimisation (SAT).

L'intérêt de la plus grande expressivité de notre modèle peut tout d'abord être vu à travers la comparaison de notre minimisation avec celle de MeMin. En effet il existe 77 cas pour lesquels MeMin n'a pas été capable de délivrer une machine aussi petite que celle obtenue par notre méthode SAT.

Il faut également mettre en avant que même dans le cas de la réduction il existe 74 cas pour lesquels la bisimulation avec spécialisation des sorties est plus efficace que MeMin au sens du nombre d'états.

L'influence de la spécialisation des sorties sur la bisimulation qui était décrite dans la section précédente est de nouveau visible ici puisque pour 229 cas cette spécialisation permet une réduction plus importante que la simple bisimulation. On voit également

	>(1)	>(2)	>(3)	>(4)	$\frac{taille}{orig}$ moy.	$\frac{taille}{orig}$ med.	$\frac{taille}{min}$ moy.	$\frac{taille}{min}$ med.	$\frac{taille}{orig}$ moy.	$\frac{taille}{orig}$ med.	$\frac{taille}{min}$ moy.	$\frac{taille}{min}$ med.
originale	73	247	204	247	1.00	1.0	4.23	1.0	1.00	1.00	9.28	1.60
(1) bisim (full)		229	168	229	0.96	1.0	1.38	1.0	0.89	1.00	1.75	1.48
(2) bisim w/ o.a. (full)		0	23	39	0.83	1.0	1.04	1.0	0.63	0.67	1.07	1.00
(3) MeMin (minimal cube)	74	0		77	0.87	1.0	1.16	1.0	0.65	0.69	1.16	1.00
(4) SAT (full)		0	0	0	0.82	1.0	1.00	1.0	0.61	0.62	1.00	1.00
	539 instances sans dépassement de temps								204 instances non-minimales sans dépassement de temps			

TABLE 7.1. – Statistiques de nos trois méthodes de réductions pour les cas liés à la SYNTCOMP. La partie la plus à gauche compte le nombre d'instances où un algorithme (y) conduit à un meilleur résultat que l'algorithme (x) ; par exemple la bisimulation avec spécialisation des sorties (2) surpasse la bisimulation standard (1) dans 229 cas. La partie du milieu présente les ratios moyens (moy.) et médians (med.) par rapport à la taille de la machine originale et la taille minimale sur l'ensemble des machines. La partie droite présente des statistiques similaires en ignorant les cas qui sont déjà minimaux.

de nouveau que l'ajout de spécialisation de sortie permet de minimiser 190 automates supplémentaires que par la simple fusion d'états équivalents.

Alors que la bisimulation avec spécialisation des sorties ne laisse que 39 machines non-minimales, on peut voir que la taille moyenne des machines résultantes qui n'étaient pas déjà minimales reste 1,07 fois plus grande que le résultat optimal. Ce gain est très important puisque les machines de départ non-minimales sont en moyenne 9,28 fois plus grandes que l'optimale même si une simple bisimulation était déjà suffisante pour réduire ce rapport à 1,75.

Puisque le but de la bisimulation avec spécialisation des sorties était de réduire la durée de réduction de la machine par rapport à une minimisation, il nous faut maintenant regarder si cette perte de qualité de résultat est compensée par un gain de temps significatif.

Dans la section 7.6.1, nous avons montré que la durée de traitement des deux minimisation est relativement proche, même si un gain est visible pour les cas où notre outil n'a pas besoin d'utiliser un SAT-solver. Nous allons donc comparer ici notre minimisation avec les deux méthodes de réduction.

Pour cela nous allons nous appuyer sur la figure 7.7. La première chose que l'on peut voir est que pour une grande partie des cas, il est possible d'obtenir un résultat minimal en moins d'un millième de seconde tant pour notre minimisation que pour les réductions. Pour les cas plus longs à traiter, commençons par ceux proches de la diagonale. Une grande partie d'entre eux concerne la bisimulation avec spécialisation des sorties. De plus il s'agit souvent de cas pour lesquels la machine de départ est déjà minimale (points pleins). Ce même phénomène est visible pour quelques cas de la bisimulation seule même si dans ce cas, elle reste plus rapide que la minimisation. À l'inverse, si on regarde les cas

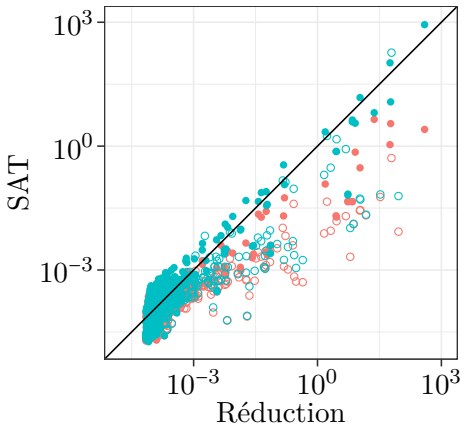


FIGURE 7.7. – Comparaison de la durée d'exécution des méthodes de réduction par rapport à notre minimisation. Les points pleins correspondent aux cas où la machine de départ est minimale. Les points rouges correspondent à la bisimulation seule alors que les points bleus correspondent à la bisimulation avec spécialisation des sorties.

pour lesquels il a fallu plus d'un dixième de seconde pour une minimisation mais où la machine de départ n'est pas minimale, alors les deux méthodes basées sur la bisimulation sont beaucoup plus rapides.

7.7. Conclusion

Nous avons donc présenté dans cette partie deux méthodes permettant de réduire la taille d'une IGMM.

La première vise à obtenir une machine minimale. Il s'agit d'une approche similaire à celle de MeMin mais utilise un modèle plus expressif pour être capable de représenter toutes les stratégies dans le cadre de la synthèse LTL. Nous avons vu que cette liberté supplémentaire permet en pratique d'obtenir de meilleurs résultats dans ce cadre qu'avec MeMin. Nous avons également vu que la meilleure utilisation de la solution partielle permet à notre implémentation de se passer de SAT-solver pour certains cas, ce qui lui procure un avantage sur MeMin.

La seconde vise à réduire le temps de traitement en n'imposant pas de minimalité de la solution. Pour cela, nous avons adapté la notion de bisimulation aux IGMM. Nous avons vu que la bisimulation avec spécialisation des sorties permet souvent l'obtention d'une machine minimale. De plus, lorsque la machine de départ n'est pas minimale, cette méthode est significativement plus rapide que lorsque l'on effectue une minimisation.

8. Autres optimisations du processus de synthèse

Nous avons jusqu'ici présenté comment `ltlsynt` transforme une formule LTL en un jeu qui est ensuite résolu pour obtenir un contrôleur. Ce chapitre sera l'occasion de décrire deux parties de `ltlsynt` présentes dans la figure 4.2 (page 43) mais pas encore évoquées : le découpage de formule LTL et la création de contrôleurs sans utiliser de jeu pour une certaine classe de formules.

Pour le découpage de formules, on se limitera à montrer l'impact lors de la SYNTCOMP de cette transformation décrite par Finkbeiner, Geier et Passing que nous n'avons fait qu'implémenter.

La construction directe de stratégie décrira comment est obtenu le contrôleur avant de justifier les contraintes sur la classe des formules considérées.

8.1. Découpage de formule LTL

8.1.1. Principe

L'idée derrière le découpage de spécification est de pouvoir travailler plus vite si on découpe la formule en sous-formules dont les contrôleurs sont ensuite regroupés pour former un contrôleur lié à la formule de départ.

Considérons par exemple une spécification décrivant que :

— une ampoule est allumée si et seulement si le capteur détecte une présence ;

— le chauffage est allumé si et seulement s'il fait froid.

Il n'y a ici aucune raison de lier le comportement de l'ampoule à la température et deux contrôleurs distincts peuvent être créés.

À l'inverse avec une spécification telle indiquant que :

— l'ampoule doit être allumée s'il fait nuit ;

— si l'ampoule est allumée, il faut fermer les volets ;

ici la position des volets dépend de l'état de l'ampoule qui dépend lui-même de s'il fait nuit ou non. Il y a donc une relation entre ces variables.

Finkbeiner et al. [28] décrivirent un moyen de décomposer une formule LTL de manière à pouvoir décomposer le problème.

Ils ont proposé deux approches. Alors qu'une des deux implique la construction d'automates et des tests d'équivalence, la seconde n'est qu'un travail autour de la formule mais qui ne couvre pas tous les cas.

Même si la grande complexité de cette construction fait que nous n'allons pas l'utiliser, nous pouvons donner l'idée derrière la version manipulant des automates.

La première étape est de créer un automate de Büchi $\mathcal{A}$ à partir de la formule de départ. Il s'agit ensuite de partitionner l'ensemble des variables de sortie de $\mathcal{A}$ en des ensembles X et Y et de créer des automates $\mathcal{A}_X$ et $\mathcal{A}_Y$ qui sont les restrictions de $\mathcal{A}$ à X et Y. Si $\mathcal{L}(\mathcal{A}_X)\|\mathcal{L}(\mathcal{A}_Y) \subseteq \mathcal{L}(\mathcal{A})$ où $\|$ est un opérateur de composition de langage, alors le découpage de $\mathcal{A}$ en ces deux automates est valide et ils peuvent être récursivement découpés.

Il a été montré [28] que cette construction permet effectivement de décomposer des spécifications mais cela a un impact significatif sur la durée de traitement. À l'inverse il est décrit que la méthode travaillant sur la formule LTL est utilisable en pratique.

Sans entrer dans les cas plus particuliers tels que les implications, il s'agit de traiter les formules de la forme $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$ et d'en extraire une partition $P_1, P_2, \ldots, P_k$ telle que deux éléments de classe différente ne partagent que des variables d'entrée.

Dans ce cas, on sait que ϕ est réalisable si et seulement si pour $i \in \{1, \ldots, k\}$, $\bigwedge_{\phi_j \in P_i} \phi_j$ est réalisable.

8.1.2. Évaluation expérimentale

Nous allons maintenant montrer l'impact de la décomposition de spécification sur le temps de traitement et la taille du contrôleur résultant. Pour cela nous mesurerons le nombre de portes ET du circuit AIG produit.

Pour ces mesures, `ltlsynt` a été exécuté pendant au plus 120 secondes. Une exécution A sera considérée comme plus lente qu'une exécution B si la durée de A est supérieure à la durée de B multipliée par 1,1 afin de réduire l'impact des légères variations de temps.

La machine utilisée est composée de :

— un processeur Intel® Core™ i7-3770 ;

— 4 × 4Go de RAM DDR3 ;

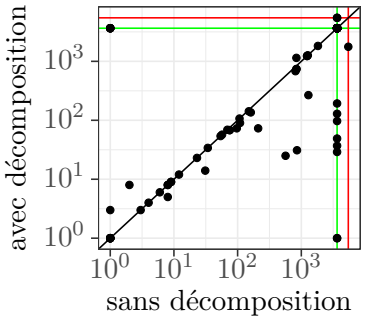
— un SSD Crucial MX500.

Nous allons nous concentrer sur les 488 formules de la SYNTCOMP pouvant être découpées. Une étude rapide de la distribution du nombre de sous-spécifications nous montre que même s'il y a entre 2 et 64 sous-spécifications, pour 89% d'entre elles il y en a au plus 10.

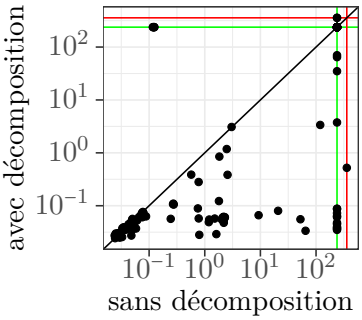
En s'appuyant sur la figure 8.1b, on peut constater qu'il y a 16 cas pour lesquels sans la décomposition `ltlsynt` n'est pas capable de donner un résultat en moins de 120 secondes alors que l'utilisation de la décomposition le permet. Cependant cette option conduit également à l'apparition de 5 cas qui ne peuvent plus être résolus.

On peut voir sur la figure 8.1a que pour 3 cas, utiliser la décomposition implique une augmentation du nombre d'états alors que pour 15 cas le contrôleur est plus petit lorsqu'elle est utilisée.

Alors que l'on vient de montrer que la décomposition permet de résoudre certains cas qui n'étaient pas résolus sans et que l'inverse est aussi vrai, nous allons maintenant étudier l'impact sur le temps de traitement.



(a) Nombre de portes ET + 1 avec et sans la décomposition



(b) Temps minimal utilisé par `ltlsynt` pour obtenir un circuit AIG avec et sans la décomposition

FIGURE 8.1. – Comparaison de l'obtention de circuit AIG avec et sans la décomposition en un ensemble de formules pouvant être décomposées. Les points sur les lignes rouges correspondent aux erreurs alors que ceux sur la ligne verte correspondent aux dépassements de temps.

La figure 8.1b nous montre qu'il n'y a que 6 cas pour lesquels la décomposition implique un temps de traitement plus long mais il ne s'agit que de cas pour lesquels il y a besoin de moins de 0,06 secondes avec la décomposition. À l'inverse, pour 43 cas, la décomposition permet un gain de temps significatif. Parmi celles-ci on peut mettre en avant le cas nommé `narylatch10` pour lequel il ne faut que 0,03 secondes lorsque l'on utilise la décomposition alors qu'il en faut 64,72 sans.

Parmi les cas qui profitent grandement de cette optimisation, on retrouve la série `shiftX` où X est un entier et de la forme $\mathsf{G}(\bigwedge_{j\in\{0,\dots,X-1\}} i_j \leftrightarrow o_{j+1 \mod X-1})$ qui est réécrite comme $\bigwedge_{j\in\{0,\dots,X-1\}} \mathsf{G}(i_j \leftrightarrow o_{j+1 \mod X-1})$, ce qui implique la création de X automates à 1 état, d'où la rapidité du traitement.

Remarque 69. *Lors de la SYNTCOMP 2021,* `ltlsynt` *a été le seul outil capable de traiter les six cas de la série* `shift` *[40]. Alors que la valeur maximale de X proposée était 64,* `Otus` *n'a pas été au-delà de $X = 32$ et $X = 16$ pour* `Strix` *alors que la limite de temps est 3600 secondes.*

À titre de comparaison, pour $X = 8192$, `ltlsynt` *n'a besoin que de 13 secondes pour construire un contrôleur (et même 9 secondes si on ajoute le bypass, discuté section 8.2.1) ou encore une seconde si on ne demande que l'existence du contrôleur.*

On peut alors conclure que même si la décomposition a un impact relativement modéré sur la qualité du résultat, il a une influence plutôt positive sur le nombre de cas qui peuvent être traités mais permet surtout une diminution du temps de traitement.

8.2. Obtention de contrôleur sans utiliser de jeu

8.2.1. Principe

Nous allons maintenant décrire une classe de formules pour lesquelles il peut être possible de savoir par une simple analyse syntaxique combinée à des opérations sur des BDD si une spécification est réalisable ou non. Si elle l'est, un contrôleur pourra être obtenu à l'aide d'une procédure que nous nommerons *bypass* qui s'affranchit de la construction d'un jeu et qui permet la construction directe d'une stratégie. Un point important est que puisque cette transformation ne peut pas être appliquée pour toutes les formules, nous ne commencerons la construction de la stratégie que si nous sommes certains que le résultat de la construction sera correct. En effet, nous verrons que pour certaines formules, il peut être possible de construire une stratégie par cette méthode mais la manière de colorer un automate peut alors influer sur la correction du résultat. Le but de cette section est de décrire cette transformation avant de montrer l'impact sur la taille du résultat mais aussi de voir si pratiquer une telle analyse syntaxique peut influer sur le temps de traitement des cas qui ne sont pas concernés par cette transformation.

Classe de formules traitée

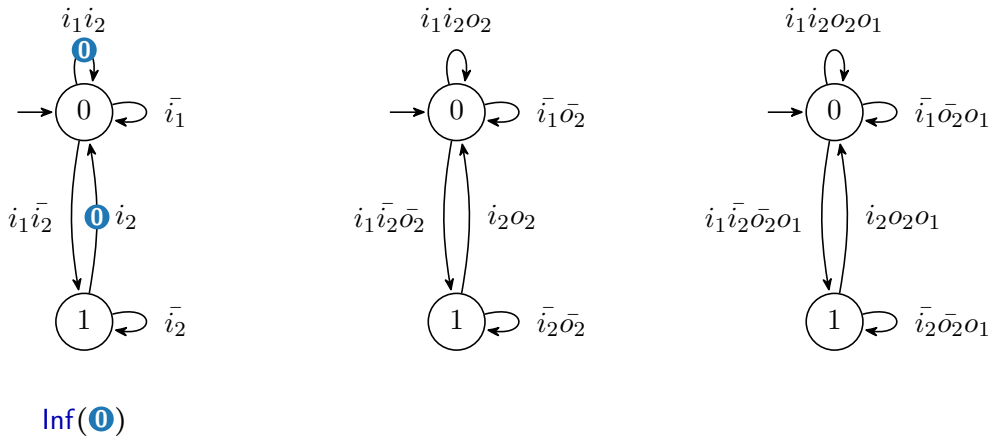
La classe des formules qui seront utilisées ici sont celles de la forme $\mathsf{G}(b_1) \wedge \underbrace{(\psi_1 \leftrightarrow \psi_2)}_{\psi}$ où b_1 est une formule Booléenne synthétisable, c'est-à-dire qu'il n'existe pas d'assignation des variables d'entrée u telle que $u \wedge b_1$ est faux et b_1 et ψ ne partagent pas de variable de sortie. De plus on doit être dans un des cas suivants :

- ψ_1 est une formule LTL contenant seulement des entrées pouvant être traduites en un automate de Büchi déterministe, ψ_2 est $\mathsf{GF}(b_2)$ où b_2 et $\neg b_2$ sont des formules Booléennes synthétisables ;
- ψ_1 est une formule LTL contenant seulement des entrées pouvant être traduites en un automate de co-Büchi déterministe, ψ_2 est $\mathsf{FG}(b_2)$ où b_2 et $\neg b_2$ sont des formules Booléennes synthétisables ;
- $b_1 \wedge b_2 \not\equiv \bot$ et $b_1 \wedge \neg b_2 \not\equiv \bot$.

Plusieurs propriétés sont utilisées pour détecter si une formule fait partie de ces cas :

- Une formule Booléenne peut être transformée en BDD et tester si elle est réalisable revient à une opération de manipulation de BDD ;
- Spot fournit une fonction `is_syntactic_recurrence` (respectivement `is_syntactic_persistence`) permettant d'affirmer qu'une formule peut être traduite en automate de Büchi (respectivement co-Büchi) même si certaines formules de récurrence (respectivement persistance) peuvent ne pas être détectées comme telles (voir page 34).

Remarque 70. *Afin de pouvoir traiter un maximum de formules détectées par cet algorithme, les règles de réécriture suivantes peuvent être appliquées :*



(a) Automate de Büchi associé à $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)$

(b) Stratégie intermédiaire obtenue à partir de l'automate de la figure 8.2a pour $(\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o_2)$ en ajoutant o_2 aux arêtes colorées et $\bar{o_2}$ aux autres

(c) Stratégie obtenue à partir de la stratégie de la figure 8.2b pour $\mathsf{G}(o_1) \wedge ((\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o_2))$ en ajoutant o_1 à toutes les arêtes

FIGURE 8.2. – Construction directe d'une stratégie pour $\mathsf{G}(o_1) \wedge ((\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o_2))$

— *Si la formule est de la forme* $\mathsf{G}(b_1) \wedge \mathsf{G}(b_2) \wedge \ldots \wedge \mathsf{G}(b_n) \wedge \psi$*, la réécrire comme* $\mathsf{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_n) \wedge \psi$ *;*

— *Si la formule est de la forme* ψ*, la réécrire comme* $\mathsf{G}(\top) \wedge \psi$ *;*

— *Si la formule est de la forme* $\mathsf{G}(b)$*, la réécrire comme* $\mathsf{G}(b) \wedge (\top \leftrightarrow \mathsf{GF}(\top))$*.*

On peut remarquer qu'utiliser la troisième réécriture implique que la formule ne correspond à aucune des règles décrites page 152 puisque $\neg b_2 = \bot$ *n'est pas réalisable. Il s'agit cependant d'un cas particulier de la construction qui sera expliqué lorsque nous décrirons pourquoi de telles restrictions sont imposées.*

Construction de la stratégie

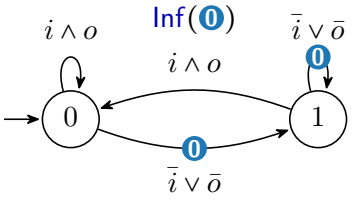
Maintenant que nous avons un moyen de savoir si une telle formule fait partie de la classe considérée, décrivons comment est construite une stratégie.

Pour cela, appuyons-nous sur l'exemple $\mathsf{G}(o_1) \wedge (\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2) \leftrightarrow \mathsf{GF}(o_2))$ où o_1 et o_2 sont les variables de sortie, ce qui correspond bien à la classe des formules étudiées.

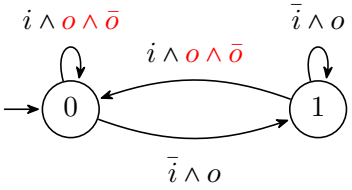
La première étape consiste à mettre de côté la partie $\mathsf{G}(o_1)$ et à traiter $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2) \leftrightarrow \mathsf{GF}(o_2)$.

Puisque $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)$ est une formule de récurrence, elle peut être traduite en automate de Büchi déterministe. Un tel automate est donné dans la figure 8.2a.

Ainsi, $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)$ est vérifié dans cet automate si et seulement si 0 est vu infiniment souvent si et seulement si b_2 est vrai infiniment souvent.



(a) Automate de Büchi associé à $\mathsf{GF}((i \wedge \bar{o}) \vee \bar{i})$



(b) Stratégie obtenue à partir de l'automate de la figure 8.3a pour la formule $\mathsf{G}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(o)$

FIGURE 8.3. – Étapes de la construction d'une fausse stratégie pour la formule $\mathsf{GF}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(o)$

L'idée est alors d'ajouter o_2 aux arêtes de l'automate de la figure 8.2a colorées par 0 et $\neg o_2$ aux autres.

Puisque toutes les exécutions de cet automate respectent $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2) \leftrightarrow \mathsf{GF}(o_2)$, les couleurs peuvent être supprimées et on obtient la stratégie de la figure 8.2b.

La dernière étape consiste alors à traiter la partie $\mathsf{G}(b_1)$. Celle-ci indique que b_1 doit être vrai à chaque instant. On ajoute donc b_1 à chaque arête de la stratégie donnée dans la figure 8.2b et on obtient la stratégie finale décrite dans la figure 8.2c.

Dans le cas des formules de la forme $\phi_1 \leftrightarrow \mathsf{FG}(\neg b_2)$, le traitement est semblable avec une traduction de ϕ_1 en automate de co-Büchi.

Remarque 71. *Puisque que la coloration d'une arête entre deux SCC n'a pas d'impact sur le langage reconnu par un automate, il est possible de mettre sur ces arêtes b_2, $\neg b_2$ ou même $b_2 \vee \neg b_2 \equiv \top$.*

Choisir de mettre $\top$ dans ce cas ajoute une liberté supplémentaire qui est à lier avec la réduction des stratégies discutée dans le chapitre précédent.

L'algorithme 7 décrit comment Spot applique cette construction tout en testant si la formule donnée a la bonne forme.

Un point important est l'appel aux lignes 9 et 13 de la fonction ToNBA permettant de créer un automate de Büchi non déterministe. En pratique, cette fonction produit souvent des automates déterministes, mais sans le garantir, même pour des récurrences. Dans ce cas nous choisissons de ne pas terminer la construction et d'utiliser la méthode générale pour trouver une stratégie.

Restrictions sur les formules traitées

Nous allons maintenant décrire pourquoi certaines restrictions sont appliquées aux formules. Nous nous appuierons sur des exemples pour montrer que si ces conditions ne sont pas respectées, alors une solution incorrecte peut être produite.

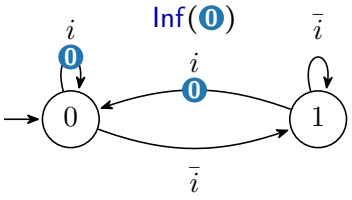
Absence de sortie dans ϕ_1 Considérons la formule $\mathsf{GF}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(o)$. Un des automates de Büchi correspondants est décrit dans la figure 8.3a.

Algorithme 7 : Création d'une stratégie pour un sous-ensemble de formules LTL

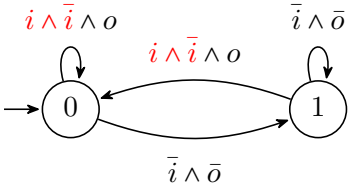
si ϕ *n'a ni la forme* $\underbrace{\mathsf{G}(b_1) \wedge (\phi_1 \leftrightarrow \mathsf{GF}(b_2))}_{\text{forme 1}}$ *où* ϕ_1 *est une récurrence syntaxique contenant seulement des entrées ni la forme* $\underbrace{\mathsf{G}(b_1) \wedge (\phi_2 \leftrightarrow \mathsf{FG}(\neg b_2))}_{\text{forme 2}}$ *où* ϕ_2 *est une persistence syntaxique avec uniquement des entrées* **alors**
  retourner non supporté
si b_1 *n'est pas réalisable* **alors**
  retourner non réalisable
si b_1 *et le reste de la formule partagent une variable de sortie* **alors**
  retourner non supporté
si b_2 *et* $\neg b_2$ *ne sont pas tous les deux réalisables* **alors**
  retourner non supporté
si ϕ *a la forme 1* **alors**
  $\mathcal{A} \leftarrow \mathsf{ToNBA}(\phi_1)$
  si $\mathcal{A}$ *n'est pas déterministe* **alors**
    retourner non supporté
sinon si ϕ *a la forme 2* **alors**
  $\mathcal{A}_{\neg\phi_2} \leftarrow \mathsf{ToNBA}(\neg\phi_2)$
  si $\mathcal{A}_{\neg\phi_2}$ *n'est pas déterministe* **alors**
    retourner non supporté
  `/* Automate de co-Büchi déterministe pour` ϕ_2 `*/`
  $\mathcal{A} \leftarrow \mathsf{Complement}(\mathcal{A}_{\neg\phi_2})$
soit $(Q, M, \Sigma, \delta, q_0, \alpha) = \mathcal{A}$
$\delta' \leftarrow \{(s, f(s, \ell, M, d), \varnothing, d) \mid (s, \ell, M, d) \in \delta\}$ où

$$f(s,\ell,M,d) = \begin{cases} \ell \wedge b_1 & \text{si } \mathsf{SccOf}(s) = \mathsf{SccOf}(d) \\ \ell \wedge b_1 \wedge b_2 & \text{si } \mathsf{SccOf}(s) \neq \mathsf{SccOf}(d) \text{ et } M = \{\textcircled{0}\} \\ \ell \wedge b_1 \wedge \neg b_2 & \text{si } \mathsf{SccOf}(s) \neq \mathsf{SccOf}(d) \text{ et } M = \varnothing \end{cases}$$

retourner $(Q, \varnothing, \Sigma, \delta', q_0, \top)$



(a) Automate de Büchi associé à $\mathsf{GF}(i)$



(b) Stratégie obtenue à partir de l'automate de la figure 8.4a pour la formule $\mathsf{GF}(i) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$

FIGURE 8.4. – Étapes de la construction d'une fausse stratégie pour $\mathsf{GF}(i) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$

Notre procédure associe $\bar{o}$ à l'arête allant de 1 à 0 et cela conduit à l'automate de la figure 8.3b où l'état 1 n'a pas d'arête sortante portant i. Il en est de même pour la boucle de l'état 0. Puisque cette machine n'est pas complète sur les entrées, ce n'est pas une stratégie.

Obligation pour b_2 et $\neg b_2$ d'être réalisables Nous allons maintenant nous appuyer sur l'exemple de la figure 8.4 pour montrer que si b_2 n'est pas réalisable, alors l'automate obtenu n'est pas toujours une stratégie valide. Le cas où $\neg b_2$ n'est pas réalisable est symétrique.

Considérons la formule $\mathsf{GF}(i) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$ où o est la seule proposition de sortie.

Ici $\bar{i} \wedge o$ n'est pas réalisable puisque si l'environnement produit i, alors il n'existe pas d'assignation de o vérifiant $\bar{i} \wedge o$.

La formule $\mathsf{GF}(i)$ peut être traduite en l'automate de la figure 8.4a. Notre procédure va associer $\bar{i} \wedge o$ aux arêtes colorées par 0. Elles portent donc maintenant la condition $i \wedge \bar{i} \wedge o \equiv \bot$, et sont supprimées. La "stratégie" obtenue décrite dans la figure 8.4b n'est pas un automate complet sur les entrées, ce n'est pas une stratégie.

Possibilité de réécrire une formule comme $\mathsf{G}(b) \wedge (\top \leftrightarrow \mathsf{GF}(\top))$ Nous avons indiqué qu'il est possible de réécrire une formule de la forme $\mathsf{G}(b)$ en $\mathsf{G}(b) \wedge (\top \leftrightarrow \mathsf{GF}(\top))$. Cependant la négation de $\top$ n'est pas réalisable.

Dans le cas de l'automate associé à $\top$, il y a un seul état portant une arête dont la condition est $\top$ et qui est colorée par 0. Il n'y a donc pas à placer $\neg b_2 \equiv \bot$ sur l'automate.

Remarque 72. *Cet automate est celui obtenu avec Spot mais n'est pas unique. On peut facilement créer un automate équivalent à deux états dont toutes les arêtes ne sont pas colorées et dans le cas ce raisonnement est faux.*

Contrainte sur le lien entre b_1 et b_2 Lors de la construction, nous avons ajouté b_2 à toutes les arêtes portant 0 et $\neg b_2$ aux autres. Ainsi, dans le cas où $b_1 \wedge b_2$ est équivalent à $\bot$, une arête serait supprimée de l'automate et il ne serait plus complet sur les entrées.

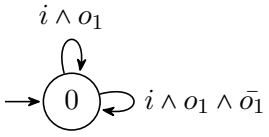


FIGURE 8.5. – Stratégie obtenue pour $\mathsf{G}(o_1) \wedge (\mathsf{GF}(i) \leftrightarrow \mathsf{GF}(o_1))$ lorsque la contrainte $b_1 \wedge b_2 \not\equiv \bot$ et $b_1 \wedge \neg b_2 \not\equiv \bot$ n'est pas respectée

Possibilité de décrire qu'une spécification n'est pas réalisable

Nous allons maintenant décrire pourquoi il peut être possible après l'analyse syntaxique de décrire qu'une formule n'est pas réalisable.

Lorsque b_1 n'est pas complet sur les entrées, cela signifie qu'il existe une suite d'affectations des variables permettant de ne pas satisfaire b_1. Puisque b_1 doit toujours être vrai, il n'est pas possible pour un contrôleur de satisfaire la condition sur une telle affectation.

8.2.2. Évaluation expérimentale

Afin d'évaluer cette méthode, nous allons construire des stratégies pour les 945 cas de la SYNTCOMP sans utiliser la décomposition. Chaque stratégie sera réduite à l'aide d'une bisimulation avec spécialisation des sorties (décrite section 7.5) et pour les cas où la méthode n'est pas appliquée, le jeu de parité est obtenu à l'aide de la méthode `lar`. Pour chaque cas deux mesures sont effectuées et la plus rapide est gardée.

Le premier élément qui peut être mis en avant est qu'il est possible d'appliquer cette construction pour 144 des 945 cas.

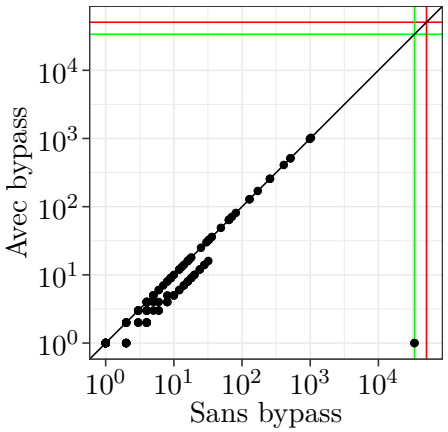
Sur ces 144, il y a 2 cas pour lesquels sans le bypass il y a un dépassement de temps mais pas avec. De plus il n'y a aucun cas qui n'est résolu que quand le bypass est désactivé.

La figure 8.6a présente une comparaison de la taille de ces stratégies pour ces 144 cas. On y voit qu'en général, la stratégie comporte peu d'états, que ce soit avec ou sans cette construction puisque 103 d'entre elles ont au plus 10 états et seulement 10 en ont au moins 100 avec au maximum 1024 états.

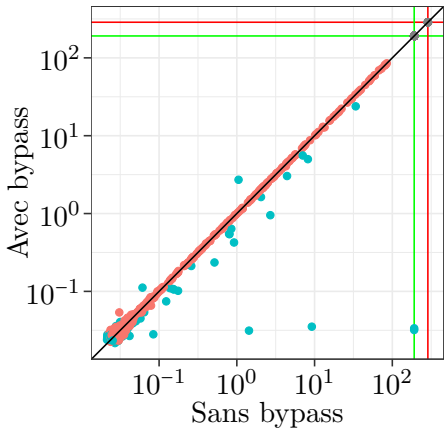
On y voit également que notre méthode ne donne jamais de stratégie plus grande que lorsque l'on passe par un jeu. En particulier pour 51 cas, nous obtenons des stratégies plus petites. Cela ne concerne cependant que les cas pour lesquels la stratégie est petite puisqu'aucune d'entre elles n'a plus de 16 états.

Parmi ces cas, on peut mettre en avant l'ensemble des cas nommés `detectorX` où X est un entier qui sont associés aux formules de la forme $\mathsf{GF}(o) \leftrightarrow \bigwedge_{i \in \{1,\dots,l\}} \mathsf{GF}(i_k)$. Sur ce type de formule notre méthode permet de créer un automate à k états où chaque état décrit quelles entrées ont été vues (à la manière de la dégénéralisation) alors qu'avec `lar`, la stratégie obtenue est toujours deux fois plus grande.

Remarque 73. *Si ACD est utilisé à la place de LAR, alors la stratégie a toujours un état de plus qu'avec le bypass. Cependant, il s'agit du cas décrit dans la section 6.4.1 où ACD est beaucoup plus lent que LAR. Ainsi pour le cas* `detector10`*,* `ltlsynt` *fournit une stratégie en 0,168s avec LAR, 0,121 avec le bypass et 10,229 avec ACD.*



(a) Nombre d'états de la stratégie avec et sans le bypass sur l'ensemble des cas pour lesquels cette construction peut être appliquée



(b) Temps minimal utilisé par `ltlsynt` avec l'algorithme `lar` et le bypass pour obtenir une stratégie. Les points bleus sont ceux pour lesquels le bypass peut être utilisé alors que ceux en gris sont ceux qui n'ont pas terminé en utilisant cette construction.

FIGURE 8.6. – Comparaison de l'obtention de stratégie entre `lar` et le bypass. Les points sur les lignes rouges correspondent aux erreurs alors que sur les lignes vertes correspondent aux dépassements de temps.

Maintenant que nous avons vu qu'utiliser cette méthode n'est jamais contre-productif en termes de qualité du résultat, il nous reste à voir si cela a un impact sur la durée de traitement.

La figure 8.6b présente la durée de traitement avec et sans utilisation du bypass. Les points bleus sont les cas où le bypass a été utilisé alors que ceux en rouge sont ceux où il a fallu passer par la méthode générale.

La première chose à mettre en avant est que lorsqu'il a été possible d'utiliser le bypass, le temps de traitement a très souvent été plus court. En particulier, il existe un ensemble de points pour lesquels le temps de traitement est quasiment nul (environ 10^{-2} secondes) avec le bypass alors que la méthode générale a besoin d'au moins une seconde ou ne termine même pas dans le temps imparti (100 secondes).

Si on considère les cas où le bypass n'a pas pu être appliqué, on voit qu'il n'y a pas d'impact significatif sur le temps de traitement, ce qui est normal puisque dans ce cas, seule une analyse syntaxique a lieu.

On peut donc conclure que cette méthode est applicable sur une partie significative des cas de la SYNTCOMP (15% d'entre eux), peut améliorer la qualité du résultat mais ne le détériore jamais et peut avoir un impact positif sur le temps de traitement sans affecter les cas qui ne peuvent pas être traités.

8.3. Conclusion

Nous avons montré qu'ajouter à `ltlsynt` un découpage en sous-spécifications permet de résoudre de traiter les cas concernés beaucoup plus rapidement que sans. Nous avons décrit une classe de formule de la SYNTCOMP mettant en avant le fait que cette étape permet de résoudre très rapidement des cas qu'aucun autre outil en compétition ne peut traiter.

Nous avons également vu qu'il est relativement souvent possible de s'affranchir de la construction usuelle lors de cette compétition. Ce bypass permet un gain léger sur la taille des contrôleurs résultants mais permet généralement un gain de temps sur les cas qui peuvent être traités sans affecter ceux qui ne le peuvent pas.

9. Conclusion

Dans ce manuscrit nous avons montré nos contributions à la transformation d'automate à condition d'Emerson-Lei en automate de parité à travers de deux procédures : `to_parity` et ACD. Nous avons également montré comment une stratégie peut être réduite en étant vue comme une machine de Mealy. Ces travaux s'inscrivent dans le cadre global de la synthèse LTL et ont été implémentés dans `ltlsynt`. Nous avons également décrit deux optimisations qui s'inscrivent également dans ce cadre : le découpage de formule et une procédure alternative de construction de stratégie pour certaines formules.

9.1. Paritisation

Les premiers résultats qui ont été présentés sont liés à la création d'un automate de parité à partir d'un automate d'Emerson-Lei.

D'un côté la procédure `to_parity` s'appuie sur la combinaison de simplifications, de procédures de transformation intermédiaires (dégénéralisation partielle), mais aussi sur des algorithmes permettant l'obtention d'automate de parité (automates parity-type, LAR ...). Des heuristiques ont été introduites telles que la recherche d'états existants ou encore l'imposition de l'ordre de déplacement pour LAR. L'implémentation de cette procédure dans Spot nous a permis de voir que ces optimisations permettent d'obtenir un automate plus petit qu'avec l'utilisation des arbres de Zielonka qui peuvent être vus comme le meilleur résultat pouvant être obtenu par un algorithme de la famille LAR seul.

De l'autre côté une nouvelle procédure nommée ACD donnant une certaine garantie d'optimalité sur la taille de l'automate résultant a été implémentée à la fois dans Spot et dans Owl. Alors que sa description était uniquement théorique jusqu'ici, ce travail a permis de montrer qu'en pratique, sa durée de traitement ainsi que la taille du résultat est comparable à ce qui est fait avec `to_parity`.

Dans les deux cas, ces deux procédures ont été intégrées à `ltlsynt`, notre outil visant à résoudre le problème de la synthèse LTL pour lequel la paritisation est une étape importante. En dehors de la configuration s'appuyant sur ACD qui n'a pas encore été testée en compétition, nous avons vu lors de la SYNTCOMP que celle s'appuyant sur `to_parity` est maintenant notre meilleure configuration, tant au niveau de la vitesse de traitement que sur la taille des contrôleurs obtenus.

9.2. Réduction de machine de Mealy

Même si l'étape de paritisation a contribué à cette réduction de taille il faut aussi mettre en avant la nouvelle étape de réduction de stratégie. Deux types de réduction ont été introduit.

Le premier vise à obtenir une stratégie de taille minimale. Des outils tels que MeMin traitent ce problème mais ce dernier s'appuie sur un modèle qui n'est pas suffisamment expressif pour décrire complètement un contrôleur. Nous l'avons donc adapté pour pouvoir l'utiliser dans le cadre de la synthèse LTL tout en y apportant diverses optimisations. Nous avons vu qu'en pratique, en plus d'être plus rapide, notre implémentation permet, grâce à cette différence de modèle, d'obtenir de meilleurs résultats.

Cependant dans les deux cas il s'agit d'algorithmes visant à résoudre un problème 2EXPTIME-complet et nous avons décidé de réduire le problème à la recherche d'un contrôleur plus petit, mais pas forcément optimal. Nous avons vu que cette seconde méthode permet en pratique une réduction très importante.

9.2.1. Autres optimisations du processus de synthèse

Alors que l'étape de paritisation et la réduction de stratégie sont deux étapes pouvant être appliquées pour traiter tous les cas dans le cadre de la synthèse LTL, nous avons également étudié les effets de deux optimisations ne concernant qu'un certain type de formules.

La première consiste à effectuer un découpage en sous-problèmes résolus indépendamment. Les contraintes qui doivent être respectées entre les différentes sous-formules font qu'il n'est pas toujours possible d'effectuer un tel découpage mais lorsque cela est possible, un gain de temps non négligeable est visible. En particulier, nous avons extrait une famille de formules pour laquelle aucun outil n'utilisant pas ce découpage n'est capable (en 2021) de produire un contrôleur.

Nous avons également montré que pour une certaine classe de formules il est possible de se passer de la construction usuelle pour créer une stratégie. Si l'impact de cette méthode sur la taille de la stratégie reste faible, il peut cependant apporter un gain de temps non-négligeable sur le temps de traitement. De plus, puisque décider si cette construction peut être fait par une analyse syntaxique de la formule, cette optimisation n'a pas d'impact significatif sur la durée de traitement des cas qui ne sont pas concernés.

9.3. Travail futur

9.3.1. Minimisation de l'ensemble des stratégies d'un jeu

Dans Spot, nous extrayons d'un jeu une stratégie et cette stratégie est minimisée.

Cependant il peut exister plusieurs stratégies pour un même jeu et ce choix de n'extraire qu'une stratégie peut ne pas être optimal.

On peut alors se dire que l'extraction d'une stratégie serait plus simple dans l'automate contenant toutes les stratégies si celui-ci est minimal. Cet automate n'est pas une

IGMM parce qu'elle introduit un non-déterminisme. Les procédures semblables à celles de MeMin ne peuvent donc pas être appliquées. Cependant les réductions basées sur la bisimulation ne différencient pas les propositions d'entrée et de sortie pour voir les machines comme des automates. Elle se base sur des inclusions de langages, ce qui permet de travailler sur ces machines non-déterministes.

9.3.2. Amélioration de l'étape de traduction

Alors que l'ancienne procédure de paritisation de `ltlsynt` nécessitait beaucoup de temps de traitement, nos nouvelles procédures ont grandement réduit la part du temps total passer à transformer un TELA en automate de parité. Une autre partie du processus est généralement celle qui prend le plus de temps : la création du TELA à partir de la formule LTL. L'outil Strix propose un découpage de formule et des procédures de combinaison d'automates de manière à obtenir directement un automate de parité [50].

Une méthode semblable pourrait être adaptée à Spot en s'appuyant notamment sur les transformations spécialisées proposées (telles que la production d'un automate déterministe faible minimal pour une obligation).

Bibliographie

[1] Andreas Abel et Jan Reineke. “MEMIN : SAT-based exact minimization of incompletely specified Mealy machines”. In : *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, p. 94-101. doi : 10.1109/ICCAD.2015.7372555.

[2] Alex Alberto et Adenilso Simao. “Iterative minimization of partial finite state machines”. In : *Open Computer Science* 3.2 (2013), p. 91-103. doi : doi:10.2478/s13537-013-0106-0. url : https://doi.org/10.2478/s13537-013-0106-0.

[3] Souheib Baarir et Alexandre Duret-Lutz. “SAT-based Minimization of Deterministic ω-Automata”. In : *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*. T. 9450. Lecture Notes in Computer Science. Springer, nov. 2015, p. 79-87. doi : 10.1007/978-3-662-48899-7_6.

[4] Tomáš Babiak et al. “Compositional Approach to Suspension and Other Improvements to LTL Translation”. In : *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)*. T. 7976. Lecture Notes in Computer Science. Springer, juill. 2013, p. 81-98. doi : 10.1007/978-3-642-39176-7_6.

[5] Tomáš Babiak et al. “The Hanoi Omega-Automata Format”. In : *Proceedings of the 27th International Conference on Computer Aided Verification (CAV'15)*. T. 9206. Lecture Notes in Computer Science. Springer, juill. 2015, p. 479-486. doi : 10.1007/978-3-319-21690-4_31.

[6] Christel Baier et al. “Generic Emptiness Check for Fun and Profit”. In : *Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis (ATVA'19)*. T. 11781. Lecture Notes in Computer Science. Springer, oct. 2019, p. 445-461. doi : 10.1007/978-3-030-31784-3_26.

[7] Dirk Beyer, Stefan Löwe et Philipp Wendler. “Reliable benchmarking : requirements and solutions”. In : *International Journal on Software Tools for Technology Transfer* 21.1 (2019), p. 1-29.

[8] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20070427*. 2007.

[9] Udi Boker. “Why These Automata Types ?” In : *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Sous la dir. de Gilles Barthe, Geoff Sutcliffe et Margus Veanes. T. 57. EPiC Series in Computing. EasyChair, 2018, p. 143-163. doi : 10.29007/c3bj. url : https://easychair.org/publications/paper/G5dD.

[10] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In : *IEEE Transactions on Computers* 35.8 (août 1986), p. 677-691.

[11] Cristian S. Calude et al. “Deciding Parity Games in Quasipolynomial Time”. In : *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2017. Montreal, Canada : Association for Computing Machinery, 2017, 252–263. isbn : 9781450345286. doi : 10.1145/3055399.3055409. url : https://doi.org/10.1145/3055399.3055409.

[12] Olivier Carton. “Chain automata”. en. In : *Theoretical Computer Science* 161.1 (juill. 1996), 191–203. issn : 0304-3975. doi : 10.1016/0304-3975(95)00103-4.

[13] Olivier Carton et Ramón Maceiras. “Computing the Rabin index of a parity automaton”. In : *Informatique théorique et applications* 33.6 (1999), p. 495-505.

[14] Antonio Casares, Thomas Colcombet et Nathanaël Fijalkow. “Optimal Transformations of Games and Automata Using Muller Conditions”. In : *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21)*. Sous la dir. de Nikhil Bansal, Emanuela Merelli et James Worrell. T. 198. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 123 :1-123 :14. doi : 10.4230/LIPIcs.ICALP.2021.123.

[15] Antonio Casares et al. “Practical Applications of the Alternating Cycle Decomposition”. In : *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science. Avr. 2022, p. 99-117. doi : 10.1007/978-3-030-99527-0_6. url : https://www.lrde.epita.fr/~frenkin/publications/tacas22/tacas22.pdf.

[16] Ivana Cerna et Radek Pelánek. “Relating Hierarchy of Temporal Properties to Model Checking”. In : *International Symposium on Mathematical Foundations of Computer Science*. T. 2747. Août 2003, p. 318-327. isbn : 978-3-540-40671-6. doi : 10.1007/978-3-540-45138-9_26.

[17] Krishnendu Chatterjee, Thomas A. Henzinger et Nir Piterman. “Generalized Parity Games”. In : *Foundations of Software Science and Computational Structures*. Sous la dir. d'Helmut Seidl. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 153-167. isbn : 978-3-540-71389-0.

[18] Christian Dax, Jochen Eisinger et Felix Klaedtke. “Mechanizing the Powerset Construction for Restricted Classes of ω-Automata”. In : *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*. Sous la dir. de Kedar S. Namjoshi et al. T. 4762. Lecture Notes in Computer Science. Springer, oct. 2007. doi : 10.1007/978-3-540-75596-8_17.

[19] Giuseppe De Giacomo et Moshe Y. Vardi. “Synthesis for LTL and LDL on Finite Traces”. In : *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina : AAAI Press, 2015, p. 1558-1564. isbn : 9781577357384.

[20] Alexandre Duret-Lutz. “LTL Translation Improvements in Spot 1.0”. In : *International Journal on Critical Computer-Based Systems* 5.1/2 (mars 2014), p. 31-54. doi : 10.1504/IJCCBS.2014.059594.

[21] Alexandre Duret-Lutz et al. “From Spot 2.0 to Spot 2.10 : What’s New ?” In : *Proceedings of the 34th International Conference on Computer Aided Verification (CAV’22)*. T. 13372. Lecture Notes in Computer Science. Springer, août 2022, p. 174-187. doi : 10.1007/978-3-031-13188-2_9.

[22] E. Allen Emerson et Chin-Laung Lei. “Modalities for Model Checking : Branching Time Logic Strikes Back”. In : *Science of Computer Programming* 8.3 (juin 1987), p. 275-306. doi : 10.1016/0167-6423(87)90036-0.

[23] E.A. Emerson et C.S. Jutla. “The complexity of tree automata and logics of programs”. In : *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*. 1988, p. 328-337. doi : 10.1109/SFCS.1988.21949.

[24] Javier Esparza, Jan Křetínský et Salomon Sickert. “One Theorem to Rule Them All : A Unified Translation of LTL into ω-Automata”. In : *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’18)*. Sous la dir. d’Anuj Dawar et Erich Grädel. ACM, 2018, p. 384-393. doi : 10.1145/3209108.3209161.

[25] Javier Esparza et al. “From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata”. In : *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’17)*. T. 10205. Lecture Notes in Computer Science. Springer-Verlag, 2017, p. 426-442. doi : 10.1007/978-3-662-54577-5_25.

[26] Kousha Etessami et Gerard J. Holzmann. “Optimizing Büchi Automata”. In : *Proceedings of the 11th International Conference on Concurrency Theory (Concur’00)*. Sous la dir. de C. Palamidessi. T. 1877. Lecture Notes in Computer Science. Pennsylvania, USA : Springer-Verlag, 2000, p. 153-167.

[27] Peter Faymonville, Bernd Finkbeiner et Leander Tentrup. “BoSy : An Experimentation Framework for Bounded Synthesis”. In : *Proceedings of CAV*. T. 10427. LNCS. Springer, 2017, p. 325-332. doi : 10.1007/978-3-319-63390-9_17.

[28] Bernd Finkbeiner, Gideon Geier et Noemi E. Passing. “Specification Decomposition for Reactive Synthesis (Full Version)”. In : *ArXiv* abs/2103.08459 (2021).

[29] Oliver Friedmann et Martin Lange. “Solving Parity Games in Practice”. In : *Automated Technology for Verification and Analysis*. Sous la dir. de Zhiming Liu et Anders P. Ravn. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 182-196. isbn : 978-3-642-04761-9.

[30] Paul GASTIN et Denis ODDOUX. "Fast LTL to Büchi Automata Translation". In : *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*. Sous la dir. de G. BERRY, H. COMON et A. FINKEL. T. 2102. Lecture Notes in Computer Science. Paris, France : Springer-Verlag, 2001, p. 53-65. DOI : 10.1007/3-540-44585-4_6.

[31] Dimitra GIANNAKOPOULOU et Flavio LERDA. "From States to Transitions : Improving Translation of LTL Formulae to Büchi Automata". In : *International Conference on Formal Techniques for Networked and Distributed Systems*. T. 2529. Sept. 2002, p. 308-326. ISBN : 978-3-540-00141-6. DOI : 10.1007/3-540-36135-9_20.

[32] Erich GRÄDEL. "Positional Determinacy of Infinite Games". In : *STACS 2004*. Sous la dir. de Volker DIEKERT et Michel HABIB. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 4-18. ISBN : 978-3-540-24749-4.

[33] Erich GRADEL et Wolfgang THOMAS. *Automata, Logics, and Infinite Games : A Guide to Current Research*. Sous la dir. d'Erich GRÄDEL, Wolfgang THOMAS et Thomas WILKE. T. 2500. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer-Verlag, 2002. ISBN : 3540003886. DOI : 10.1007/3-540-36387-4.

[34] M.R. HENZINGER, T.A. HENZINGER et P.W. KOPKE. "Computing simulations on finite and infinite graphs". In : *Proceedings of the 36th Symposium on Foundations of Computer Science*. 1995, p. 453-462. DOI : 10.1109/SFCS.1995.492576.

[35] Thomas HENZINGER et Nir PITERMAN. "Solving Games Without Determinization". In : *CSL*. 2006, p. 395-410. DOI : 10.1007/11874683_26.

[36] John HOPCROFT. "An n log n algorithm for minimizing states in a finite automaton". In : *Theory of Machines and Computations*. Sous la dir. de Zvi KOHAVI et Azaria PAZ. Academic Press, 1971, p. 189-196. ISBN : 978-0-12-417750-5. DOI : https://doi.org/10.1016/B978-0-12-417750-5.50022-1. URL : https://www.sciencedirect.com/science/article/pii/B9780124177505500221.

[37] "IEC 62531 Ed. 1 (2007-11) (IEEE Std 1850-2005) : Standard for Property Specification Language (PSL)". In : *IEC 62531 :2007 (E)* (déc. 2007), p. 1-152. DOI : 10.1109/IEEESTD.2007.4408637.

[38] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In : *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (fév. 2018), p. 1-1315. DOI : 10.1109/IEEESTD.2018.8299595.

[39] Swen JACOBS et al. "The 5th Reactive Synthesis Competition (SYNTCOMP 2018) : Benchmarks, Participants & Results". In : *CoRR* abs/1904.07736 (avr. 2019). arXiv : 1904.07736. URL : http://arxiv.org/abs/1904.07736.

[40] Swen JACOBS et al. "The Reactive Synthesis Competition (SYNTCOMP) : 2018-2021". In : *International Journal on Software Tools for Technology Transfer* (2022). submitted. URL : https://ui.adsabs.harvard.edu/link_gateway/2022arXiv220600251J/arxiv:2206.00251.

[41] Timothy KAM et al. "A fully implicit algorithm for exact state minimization". In : *31st Design Automation Conference*. IEEE. 1994, p. 684-690.

[42] Zuzana KOMÁRKOVÁ et Jan KŘETÍNSKÝ. "Rabinizer 3 : Safraless Translation of LTL to Small Deterministic Automata". In : *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)*. Sous la dir. de Franck CASSEZ et Jean-François RASKIN. Cham : Springer International Publishing, 2014, p. 235-241. DOI : 10.1007/978-3-319-11936-6_17.

[43] Jan KŘETÍNSKÝ et al. "Index Appearance Record for Transforming Rabin Automata into Parity Automata". In : *Proceedings of the 23st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Sous la dir. d'Axel LEGAY et Tiziana MARGARIA. T. 10205.I. Lecture Notes in Computer Science. 2017, p. 443-460. DOI : 10.1007/978-3-662-54577-5_26.

[44] Sriram C. KRISHNAN, Anuj PURI et Robert K. BRAYTON. "Deterministic ω-Automata Vis-a-Vis Deterministic Büchi Automata". In : *Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC'94)*. ISAAC '94. Berlin, Heidelberg : Springer-Verlag, 1994, 378–386. ISBN : 3540583254.

[45] O. KUPFERMAN. "Avoiding Determinization". In : *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. 2006, p. 243-254. DOI : 10.1109/LICS.2006.15.

[46] Jan KŘETÍNSKÝ et al. "Index Appearance Record with preorders". In : *Acta Informatica* (2021). DOI : 10.1007/s00236-021-00412-y.

[47] Christof LÖDING. "Methods for the Transformation of ω-Automata : Complexity and Connection to Second Order Logic". Diploma Thesis. Institute of Computer Science et Applied Mathematics, 1998.

[48] Christof LÖDING. "Optimal Bounds for Transformations of ω-Automata". In : *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99)*. T. 1738. Lecture Notes in Computer Science. Springer, 1999, p. 97-109. DOI : 10.1007/3-540-46691-6_8.

[49] Christof LÖDING et Wolfgang THOMAS. "Alternating Automata and Logics over Infinite Words". In : *Theoretical Computer Science : Exploring New Frontiers of Theoretical Informatics*. Sous la dir. de Jan van LEEUWEN et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 521-535. ISBN : 978-3-540-44929-4.

[50] Michael LUTTENBERGER, Philipp J. MEYER et Salomon SICKERT. "Practical synthesis of reactive systems from LTL specifications via parity games". In : *Acta Informatica* 57.1-2 (2020), p. 3-36. DOI : 10.1007/s00236-019-00349-3. URL : https://doi.org/10.1007/s00236-019-00349-3.

[51] Juraj MAJOR et al. "ltl3tela : LTL to Small Deterministic or Nondeterministic Emerson-Lei Automata". In : *Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis (ATVA'19)*. Sous la dir. d'Yu-Fang CHEN, Chih-Hong CHENG et Javier ESPARZA. T. 11781. Lecture Notes in

Computer Science. Springer, 2019, p. 357-365. DOI : 10.1007/978-3-030-31784-3_21.

[52] Zohar MANNA et Amir PNUELI. "A Hierarchy of Temporal Properties (Invited Paper, 1989)". In : *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*. PODC '90. Quebec City, Quebec, Canada : Association for Computing Machinery, 1990, 377–410. ISBN : 089791404X. DOI : 10.1145/93385.93442. URL : https://doi.org/10.1145/93385.93442.

[53] Nicolas MARKEY. "Temporal Logic with Past is Exponentially More Succinct". In : *EATCS Bulletin* 79 (fév. 2003), p. 122-128.

[54] Donald A. MARTIN. "Borel Determinacy". In : *Annals of Mathematics* 102.2 (1975), p. 363-371. ISSN : 0003486X. URL : http://www.jstor.org/stable/1971035 (visité le 16/05/2022).

[55] George H. MEALY. "A method for synthesizing sequential circuits". In : *The Bell System Technical Journal* 34.5 (1955), p. 1045-1079. DOI : 10.1002/j.1538-7305.1955.tb03788.x.

[56] Shin-ichi MINATO. "Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams". In : *Proceedings of the third Synthesis and Simulation and Meeting International Interchange workshop (SASIMI'92)*. Kobe, Japan, avr. 1992, p. 64-73.

[57] David MÜLLER et Salomon SICKERT. "LTL to Deterministic Emerson-Lei Automata". In : *Proceedings of the Eighth International Symposium on Games, Automata, Logics and Formal Verification (GandALF'17)*. Sous la dir. de Patricia BOUYER, Andrea ORLANDINI et Pierluigi San PIETRO. T. 256. EPTCS. Sept. 2017, p. 180-194. DOI : 10.4204/EPTCS.256.13.

[58] J.M. PENA et A.L. OLIVEIRA. "A new algorithm for exact reduction of incompletely specified finite state machines". In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.11 (1999), p. 1619-1632. DOI : 10.1109/43.806807.

[59] Charles P. PFLEEGER. "State Reduction in Incompletely Specified Finite-State Machines". In : *IEEE Transactions on Computers* C-22.12 (1973), p. 1099-1102. DOI : 10.1109/T-C.1973.223655.

[60] N. PITERMAN et A. PNUELI. "Faster Solutions of Rabin and Streett Games". In : *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. 2006, p. 275-284. DOI : 10.1109/LICS.2006.23.

[61] Nir PITERMAN. "From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata". In : *LICS*. 2006, p. 255-264. DOI : 10.1109/LICS.2006.28.

[62] A. Pnueli et R. Rosner. “On the Synthesis of a Reactive Module”. In : *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA : Association for Computing Machinery, 1989, 179–190. isbn : 0897912942. doi : 10.1145/75277.75293. url : https://doi.org/10.1145/75277.75293.

[63] Amir Pnueli. “The temporal logic of programs”. In : *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, p. 46-57. doi : 10.1109/SFCS.1977.32.

[64] Roman Redziejowski. “An improved construction of deterministic omega-automaton using derivatives”. In : *Fundamenta Informaticae* 119.3-4 (2012), p. 393-406. doi : 10.3233/FI-2012-744.

[65] Florian Renkin, Alexandre Duret-Lutz et Adrien Pommellet. “Practical “Paritizing” of Emerson-Lei Automata”. In : *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA’20)*. T. 12302. Lecture Notes in Computer Science. Springer, oct. 2020, p. 127-143. doi : 10.1007/978-3-030-59152-6_7.

[66] Florian Renkin et al. “Dissecting `ltlsynt`”. Avr. 2022. url : https://www.lrde.epita.fr/~frenkin/publications/fmsd22/fmsd22.pdf.

[67] Florian Renkin et al. “Effective Reductions of Mealy Machines”. In : *Proceedings of the 42nd International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE’22)*. Lecture Notes in Computer Science. To appear. Springer, juin 2022. url : https://www.lrde.epita.fr/~frenkin/publications/forte22/forte22.pdf.

[68] June-Kyung Rho et al. “Exact and heuristic algorithms for the minimization of incompletely specified state machines”. In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.2 (1994), p. 167-177. doi : 10.1109/43.259940.

[69] S. Safra et M. Y. Vardi. “On ω-automata and temporal logic”. In : *Proceedings of the twenty-first annual ACM symposium on Theory of computing - STOC ’89*. ACM Press, 1989, 127–137. isbn : 978-0-89791-307-2. doi : 10.1145/73007.73019. url : http://portal.acm.org/citation.cfm?doid=73007.73019.

[70] Schmuel Safra. “On the Complexity of ω-Automata”. In : *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS’88)*. 1988, p. 319-327. doi : 10.1109/SFCS.1988.21948.

[71] Martin Shubik. *Game Theory in the Social Sciences : Concepts and Solutions*. MIT press, 1982. isbn : 0262690918.

[72] Moshe Y. Vardi et Pierre Wolper. “An Automata-Theoretic Approach to Automatic Program Verification”. In : *Proceedings of the 1st Symposium on Logic in Computer Science (LICS’86)*. IEEE Computer Society Press, juin 1986, p. 332-344.

[73] Mark WALKER et John WOODERS. "Minimax Play at Wimbledon". In : *The American Economic Review* 91.5 (2001), p. 1521-1538. ISSN : 00028282. URL : http://www.jstor.org/stable/2677937.

[74] Saeyang YANG. *Logic synthesis and optimization benchmarks user guide : version 3.0*. Citeseer, 1991.

[75] Ernst ZERMELO. "Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels". In : *Proceedings of the Fifth International Congress of Mathematicians*. T. 2. Cambridge : Cambridge University Press, 1913, p. 501-504.

A. Lien entre machine de Mealy et circuit AIG

Nous allons ici voir le lien entre taille d'IGMM et taille du circuit AIG obtenu en sortie. Il ne s'agira que d'une vision basique permettant au lecteur d'avoir une idée du processus de conversion d'une représentation à l'autre.

Cette description fait l'objet d'un article soumis pour une édition spéciale du journal FMSD [66].

Pour rappel un AIG correspond à un circuit ne portant que des portes ET, des négations et des bascules.

Il convient de rappeler que la sortie d'une IGMM est décrite par une fonction $\lambda : Q \times \mathbb{B}^I \to 2^{\mathbb{B}^O} \setminus \{\varnothing\}$ et elle associe donc à une entrée un ensemble de sorties possibles. Un AIG ne peut quant à lui qu'associer à une entrée une unique sortie. Ce dernier devra donc être une spécialisation de l'IGMM.

Pour résoudre ce choix, nous utiliserons une fonction ChooseOneValuation.

Remarque 74. *Ici cette fonction va être une boite noire mais « Dissecting `ltlsynt` » [66] présentera une heuristique permettant de décider quelle sortie doit être choisie.*

Pour encoder une machine $\mathcal{M} = (I, O, Q, q_{\text{init}}, \delta, \lambda)$ comme un AIG de manière naïve, supposons sans perte de généralité que chaque état peut être associé à un unique nombre dans l'ensemble $\{0, \ldots, |Q| - 1\}$ où l'état initial q_{init} correspond à 0.

Nous utiliserons un ensemble de $N_l = \lceil \log_2(|Q|) \rceil$ bascules permettant de mémoriser l'état actuel de la machine. Notons $\mathsf{BinaryEnc}(q)$ le tableau de N_ℓ valeurs Booléennes correspondant à l'encodage binaire de q.

La première étape est de transformer l'IGMM en une représentation symbolique spécifique exprimée comme des fonctions Booléennes à l'aide de diagrammes de décision binaire (BDD) [10]. Un tel BDD est créé pour chaque signal de sortie et chaque bascule.

Pour une bascule $\ell \in \{\ell_0, \ldots, \ell_{N_l - 1}\}$, on note v_ℓ sa valeur actuelle et f^ℓ l'encodage à l'aide d'un BDD de la fonction calculant sa valeur suivante. De manière analogue, pour une sortie $o \in O$, f^o est l'encodage sous forme de BDD de la fonction associée à ce signal. Toutes ces fonctions f^ℓ et f^o sont des fonctions de la valeur courante des bascules (v_{ℓ_j}) et des signaux d'entrée $i \in I$.

L'algorithme 8 montre la manière dont ces fonctions sont calculées à partir d'une IGMM.

Ces BDD peuvent alors être traduits en AIG à l'aide de l'*If-Then-Else normal form* (ITE, aussi connue sous le nom de théorème d'expansion de Boole) basée sur l'identité $f = (v \wedge f_v) \vee (\bar{v} \wedge v_{\bar{b}})$. Dans notre cas, f est soit une fonction de sortie f^o ou une fonction de bascule f^ℓ, v correspond à une entrée ou une bascule et $f_v / f_{\bar{v}}$ est f où l'argument

Algorithme 8 : ToSymbolicMealy

Entrée : IGMM $M = (I, O, Q, q_{init}, \delta, \lambda)$
Sortie : Fonctions Booléennes pour les sorties et bascules

$\forall j \in \{0, \dots, N_\ell - 1\}$: $f^{\ell_j} \leftarrow \bot$
$\forall o \in O$: $f^o \leftarrow \bot$
pour $q \in Q$ **faire**
  $f^q = \top$
  `/* Encode` q `comme une conjonction de bascules */`
  pour $j \in \{0, \dots, N_\ell - 1\}$ **faire**
    si BinaryEnc$(q)[j]$ **alors**
      $f^q \leftarrow f^q \wedge v_{\ell_j}$
    sinon
      $f^q \leftarrow f^q \wedge \bar{v}_{\ell_j}$
  `/* Met à jour les fonctions pour chaque entrée */`
  pour $u \in \mathbb{B}^I$ **faire**
    si $\delta(q, u)$ ***est*** *non défini* **alors**
      continue
    $q' = \delta(q, u)$
    $v =$ ChooseOneValuation$(\lambda(q, u))$
    pour $j \in \{0, \dots, N_\ell - 1\}$ **faire**
      si BinaryEnc$(q')[j]$ **alors**
        $f^{\ell_j} \leftarrow f^{\ell_j} \vee (f^q \wedge u)$
      pour $o \in O$ **faire**
        si $v \to o$ **alors**
          $f^o \leftarrow f^o \vee (f^q \wedge u)$
retourner $\{f^o \mid o \in O\}, \{f^\ell \mid \ell \in \{\ell_0, \dots, \ell_{N_\ell - 1}\}\}$

v est $\top/\bot$. Cette application récursive va créer de manière naturelle des arbres portant trois portes ET. Un exemple est donné dans la figure A.1.

Nous avons donc vu qu'en encodant en AIG, le nombre de bascules dépend du nombre d'états ($\lceil \log_2(|Q|) \rceil$). De plus les fonctions f^ℓ et f^o encodant la gestion des bascules et des sorties dépend de ces bascules. On peut donc déjà voir que la taille de l'automate influe sur la taille du circuit.

Même si cela ne sera pas discuté ici, l'article soumis pour une édition spéciale du journal FMSD [66] décrira également comment nous choisissons les sorties de l'IGMM pour limiter le nombre de portes utilisées pour encoder les fonctions de sortie.

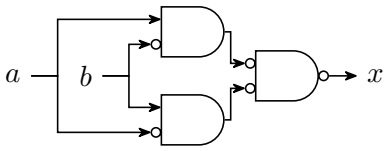


FIGURE A.1. – Circuit associant à x le résultat du OU exclusif entre les entrées a et b. La fonction Booléenne devient alors $f^x = (a \wedge \bar{b}) \vee (\bar{a} \wedge b)$. Cette formule est déjà sous forme ITE et le circuit AIG est décrit ici.

B. Travail autour de Spot

Comme nous l'avons vu, l'ensemble de ce travail a été intégré à Spot. Ce chapitre a pour but de donner quelques informations sur mon apport à cette bibliothèque durant cette thèse.

B.1. Procédure de paritisation

B.1.1. Méthodes présentes avant le début de la thèse

Avant le début de cette thèse, deux méthodes de paritisation basées autour de LAR étaient présentes dans Spot.

La première était `to_parity`, devenue `to_parity_old` qui applique CAR sur un automate quelconque mais aucune optimisation n'y est utilisée.

La seconde était `iar` (et sa variante `iar_maybe`) qui applique IAR sur un automate de Rabin ou de Streett. Dans cette fonction, plusieurs optimisations étaient présentes et ont été reprises dans `to_parity` :

— le découpage du problème en la paritisation de plusieurs SCC même si aucune simplification n'est appliquée ;

— la conservation des SCC terminales ;

— la recherche d'état existant.

Une adaptation de la procédure de détection des automates de Rabin qui sont Büchi-types de Krishnan et al. était présente dans Spot et a été généralisée pour produire des automates parity-type à partir d'automates à condition quelconque.

B.1.2. Nouvelles procédures implémentées durant cette thèse

Plusieurs procédures liées au domaine de la paritisation ont été introduites.

Commençons par évoquer `to_parity`, la méthode générale décrite dans le chapitre 5. Elle s'appuie sur différentes procédures.

Par exemple, `parity_type_to_parity` permettant de transformer un automate parity-type en automate de parité ainsi que `buchi_type_to_buchi` qui est sa restriction aux automates de Büchi et `co_buchi_type_to_co_buchi` qui est sa restriction aux automates de co-Büchi (sec. 5.10).

Il y a également la fonction `propagate_marks_here` permettant de propager les couleurs dans un automate (sec. 5.11.7).

Enfin, la fonction `partial_degeneralize` est liée à la dégénéralisation partielle (voir sec. 5.8).

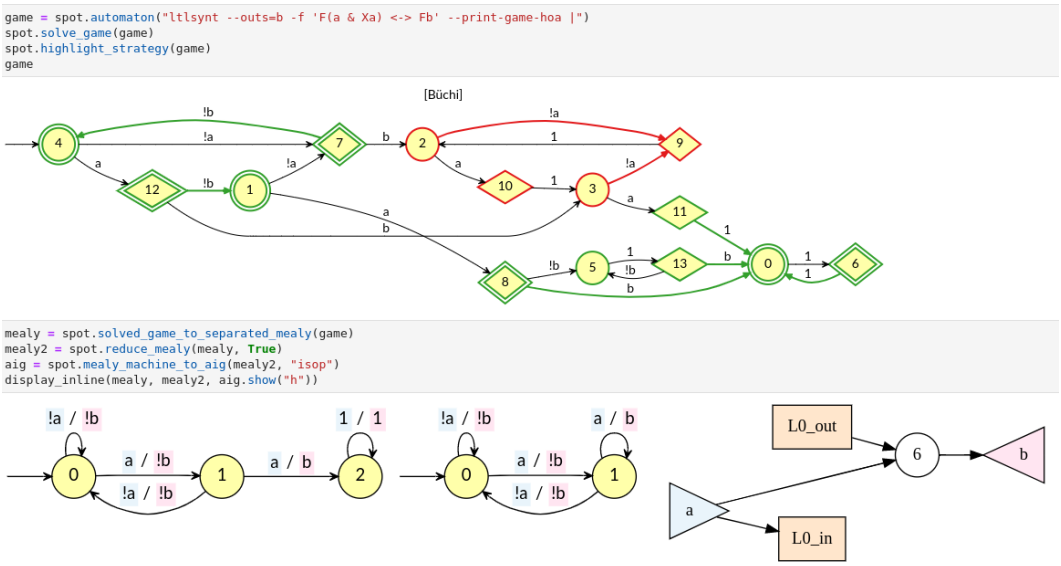


FIGURE B.1. – Notebook montrant différentes étapes de la création d'un circuit AIG à partir d'une spécification

En parallèle, Alexandre Duret-Lutz a implémenté la construction d'automate de parité à partir d'arbre de Zielonka (classe `zielonka_tree` de Spot) ainsi qu'avec ACD (`acd_transform` pour la version où les arêtes sont colorées et `acd_transform_sbacc` pour la version où les états sont colorés). Ma participation à cet effort a été de soulever quelques bugs et conduire des campagnes de test.

B.2. Machines utilisées durant la synthèse

Avant le début de cette thèse, les notions de jeu et machine de Mealy incomplètement spécifiée n'étaient utilisées que dans `ltlsynt` mais nous avons modifié la structure du programme pour pouvoir exposer ces représentations. Il s'agit d'un travail conjoint avec Philipp Schlehuber-Caissier et Alexandre Duret-Lutz.

Avant, `ltlsynt` simulait les jeux en utilisant localement un vecteur de Booléen indiquant le propriétaire d'un état. Maintenant, ce vecteur est une propriété nommée liée à l'automate. Il en est de même pour l'association à un jeu des régions gagnantes. Les jeux sont devenus des objets utilisables directement dans Spot.

Une fois la stratégie obtenue, nous la représentons maintenant comme une machine de Mealy incomplètement spécifiée. Il s'agit d'un automate dont la condition d'acceptation est $\top$ et portant la propriété `synthesis-outputs` décrivant quelles sont les variables de sortie.

Une classe `AIG` a également été introduite pour pouvoir manipuler ces circuits.

Toutes ces modifications permettent maintenant de montrer à travers des notebooks comme celui de la figure B.1 les étapes de la synthèse LTL.

C. Minimisation d'IGMM par MeMin

Nous allons décrire ici MeMin [1]. Il s'agit d'un outil permettant de réduire une IGMM.

Abel et Reineke utilisent un modèle d'IGMM différent de celui décrit dans la définition 69 (page 129), qui est moins expressif que celui que nous utilisons, comme indiqué dans la section 7.2.

Définition 78 (Machine de Mealy incomplètement spécifiée, [1])**.** *Une machine de Mealy incomplètement spécifiée est un tuple* $\mathcal{M} = (I, O, Q, q_r, \delta, \lambda)$ *tel que*

- I *est un ensemble non vide d'entrées ;*
- O *est un ensemble non vide de sorties ;*
- Q *est un ensemble non vide d'états ;*
- $q_r \in Q \cup \{\perp\}$ *est un état initial, potentiellement non spécifié ;*
- $\delta : (Q, I) \to Q \cup \{\phi\}$ *est la fonction de transition où* ϕ *indique une destination non spécifiée ;*
- $\lambda : (Q, I) \to O \cup \{\epsilon\}$ *est la fonction de sortie où* ϵ *indique une sortie non spécifiée.*

Pour ces machines, l'idée de variation est présente sous le nom de *compatible*, alors que la spécialisation est présente sous le terme *cover*.

Définition 79 (Classe de compatibilité, [1])**.** *Pour une IGMM* $\mathcal{M} = (I, O, Q, q_r, \delta, \lambda)$*, une classe de compatibilité est un ensemble* $C \subseteq Q$ *tel que ses éléments sont les variations les uns des autres.*

Définition 80 (Fonction successeur, [1])**.** *Pour une classe de compatibilité* $C = \{q_0, \ldots, q_n\}$ *et une entrée* a*, on définit la fonction successeur comme*

$$\mathrm{Succ}(C, a) = \bigcup_{0 \leq j \leq n} \{\delta(q_j, a) \mid \delta(q_j, a) \neq \varnothing\}$$

c'est-à-dire les états accessibles depuis un des états de C *en lisant* a*.*

Définition 81 (Ensemble de classes de compatibilité clos, [1])**.** *Un ensemble* $S = \{C_1, \ldots, C_n\}$ *de classes de compatibilité est clos si pour toute classe* $C_j \in S$ *et toute entrée* $a \in I$ *il existe une classe* $C_k \in S$ *telle que* $\mathrm{Succ}(C_j) \subseteq C_k$*.*

Définition 82 (Ensemble de classes de compatibilité couvrant une IGMM, [1])**.** *Un ensemble de classes de compatibilité couvre une IGMM* $\mathcal{M}$ *si tout état de* $\mathcal{M}$ *est contenu dans au moins une classe.*

Enfin, ils décrivent la machine minimale à partir de ce théorème :

Théorème 12 (Construction d'une IGMM minimale, [1]). *À partir d'un ensemble clos* $S = \{C_1, \ldots, C_n\}$ *de classes de compatibilité avec le nombre minimal de classes permettant de couvrir une IGMM* $\mathcal{M} = (I, O, Q, q_r, \delta, \lambda)$ *on peut créer une IGMM* $\mathcal{M}' = (I, O, Q', q'_r, \delta', \lambda')$ *qui est une spécialisation de* $\mathcal{M}$ *de la manière suivante :*

- $Q' = S$;
- $q'_r = C_j$ *pour un certain* j *tel que* $q_r \in C_j$ *si* q_r *est défini,* $\perp$ *sinon ;*
- $\delta'(C_j, a) = \phi$ *si* $\mathrm{Succ}(C_j, a) = \varnothing$ *et sinon,* $\delta'(C_j, a) = C_k$ *pour un* k *tel que* $\mathrm{Succ}(C_j, a) \subseteq C_k$;
- $\lambda'(C_j, a) = o$ *si* $\lambda(q, a) = o$ *pour un certain* $q \in C_j$ *et* $\lambda'(C_j, a) = \epsilon$ *sinon.*

La première étape de l'algorithme de Abel et Reineke est de calculer un ensemble d'états qui ne sont pas les variations les uns des autres.

L'idée est que si n tels états sont trouvés, alors la solution devra avoir au moins n états puisque le théorème 12 indique que les états sont composés de classes de compatibilité.

Une fois cette borne inférieure trouvée, le problème de minimisation est encodé sous la forme d'un problème SAT. Ce problème est satisfaisable si et seulement si, pour une taille k fixée, une machine minimale de taille k existe.

Pour cela, des variables de la forme $s_{q,i}$ où q est un état et i un indice de classe sont introduites.

La condition de couverture est alors décrite pour chaque état q par la formule

$$s_{q,0} \vee s_{q,1} \vee \ldots \vee s_{q,n-1}$$

La condition de compatibilité est elle décrite par

$$\bigwedge_{\substack{q' \in \mathrm{Inc}(q) \\ q' > q}} (\neg s_{q,i} \vee \neg s_{q',i})$$

où $\mathrm{Inc}(q)$ désigne l'ensemble des états incompatibles avec q.

Enfin, la condition de clôture est décrite par la contrainte

$$\bigvee_{0 \leq j < n} \left(\bigwedge_{q \in Q} (\neg s_{q,i} \vee s_{q',j}) \right)$$

où q' est le successeur de x pour une entrée a.