Table des matières

Table des figures

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Liste des tableaux

Rapport-Gratuit.com

Liste des algorithmes

Introduction

Un système (véhicule, instrument) est dit *autonome* s'il n'est pas contrôlé par un opérateur humain. Ses actions sont alors gérées par un programme interne, qui ne peut être modifié pendant son fonctionnement ; il s'agit d'un programme *embarqué*, devant permettre au système de « prendre lui-même des décisions ». Les systèmes embarqués font généralement l'objet de fortes limitations (par exemple au niveau de leur prix, poids ou puissance) qui dépendent de la tâche qu'ils doivent accomplir ; ils n'ont par conséquent que peu de ressources à leur disposition, en termes d'*espace mémoire* et de *puissance de calcul*. Malgré cela, ils exigent généralement une très forte *réactivité*. Ces contraintes sont particulièrement significatives dans le cadre des systèmes embarqués contrôlant des systèmes spatiaux et aéronautiques, tels des drones ou des satellites : leurs ressources peuvent être drastiquement limitées, et leur besoin en réactivité crucial.

La prise de décision est l'un des objectifs du domaine de l'intelligence artificielle que l'on nomme *planification*. Pour produire des décisions adaptées à la situation courante, le programme doit résoudre un *problème de planification*. Des outils ont été développés pour résoudre de tels problèmes ; grâce à eux, le système peut calculer des décisions convenables. Cependant, dans le cas général, les problèmes de planification sont difficiles à résoudre — ils ont une haute complexité algorithmique. En conséquence, prendre des décisions en résolvant des problèmes de planification prend du temps, et ce en particulier sur les systèmes autonomes, qui ne disposent que d'une puissance limitée. Il est donc impossible de leur assurer une bonne réactivité si le problème est résolu *en ligne*, c'est-à-dire à chaque fois qu'une décision doit être prise, en n'utilisant que la puissance et la mémoire du système.

Une façon de régler ce problème est de calculer les décisions *hors ligne*, avant la mise en situation du système. Pour ce faire, on anticipe toutes les situations possibles et on résout le problème pour chacune d'elles ; on peut alors fournir au système une table de décision, contenant un ensemble de « règles de décision » de la forme « dans *telle* situation, prends *telle* décision ». Cela garantit une réactivité maximale au système ; en effet l'exploitation en ligne de ce genre de table ne requiert qu'une puissance très limitée.

Néanmoins, ce gain en réactivité peut nécessiter une forte hausse de l'espace

mémoire nécessaire : la prise de décision dépend de nombreux paramètres, parmi lesquels des mesures, la position courante (supposée) du système, l'état des composants, les données courantes, les objectifs courants, et même, dans certains cas, les valeurs précédentes de tous ces paramètres. Ceci conduit à un nombre astronomique de situations possibles, et donc à une table de décision de taille elle aussi astronomique — alors même que l'espace mémoire disponible à bord est fortement limité.

On voit qu'il est nécessaire de trouver un compromis entre réactivité et compacité. La *compilation de connaissances* constitue l'un des moyens d'atteindre un tel compromis : cette discipline a pour objectif d'étudier comment *traduire* un problème hors ligne pour faciliter sa résolution en ligne. Elle examine dans quelle mesure certains *langages-cibles*, vers lesquels les problèmes sont *compilés*, permettent de rendre les opérations en ligne faisables tout en maintenant autant que possible la compacité de la représentation.

Les techniques de compilation de connaissances ont démontré leur utilité pour diverses applications, dont la planification ; le but de cette thèse était d'étudier la possibilité d'appliquer avantageusement ces techniques à des problèmes de planification *réalistes* portant sur des systèmes aéronautiques et spatiaux. Ayant considéré plusieurs problèmes réels, nous avons notamment remarqué que les langages-cibles utilisés dans les planificateurs actuels ne permettent pas de manipuler directement certains aspects de nos problèmes, à savoir les variables à domaines continus. Nous avons orienté notre travail vers l'application de techniques de la littérature à des langages-cibles plus expressifs.

Cette thèse est divisée en trois parties. La première développe le *contexte* de ce travail, présentant plus en détail la compilation de connaissances [chapitre 1] et la planification [chapitre 2]. Elle explique ensuite [chapitre 3] les raisons pour lesquels nous nous sommes dirigés vers l'étude de nouveaux langages-cibles de compilation. La deuxième partie traite des *automates à intervalles*, l'un des langages que nous avons définis ; elle présente certains aspects théoriques de ce langage [chapitre 4], explique comment il est possible en pratique de construire de tels automates [chapitre 5], et fournit des résultats expérimentaux [chapitre 6]. La troisième partie porte sur un autre de nos langages-cibles, celui des *set-labeled diagrams*. Elle se décompose de la même façon que la deuxième, commençant par des définitions et des propriétés générales du langage [chapitre 7], puis présentant notre algorithme de compilation [chapitre 8], et se terminant par des résultats expérimentaux [chapitre 9].

Première partie

Contexte

CHAPITRE

1

Compilation de connaissances

Ce chapitre détaille le principal objet de ce travail, la compilation de connaissances. Pour résumer, la compilation de connaissances consiste à transformer un problème hors ligne de manière à faciliter sa résolution en ligne. Cette transformation est en fait considérée comme une *traduction* du problème, depuis le *langage* original dans lequel il est décrit, vers un *langage-cible* ayant de bonnes propriétés — celles de permettre au problème d'être soluble « facilement » tout en restant aussi compact que possible.

Après avoir présenté et illustré les concepts sur lesquels se base de la compilation de connaissances [§ 1.1], nous définissons formellement les notions relatives aux *langages* [§ 1.2]. Un certain nombre de langages issus de la littérature sont ensuite définis [§ 1.3] ; ils appartiennent tous à la classe particulière des langages de graphes booléens, sur laquelle nous nous sommes concentrés durant la thèse. La section suivante détaille les propriétés de cette classe, et rappelle les résultats théoriques connus à son propos [§ 1.4]. La dernière section du chapitre est dédiée à l'étape de compilation proprement dite [§ 1.5].

1.1 Présentation

1.1.1 Concepts et historique

La compilation de connaissances peut être vue comme une forme de « factorisation », la phase hors ligne étant dédiée à l'exécution de calculs à la fois *difficiles* et *communs à plusieurs opérations en ligne*. Cette idée n'est pas nouvelle, comme l'illustre l'exemple des tables de logarithmes [Mar08], qui étaient utilisées à une époque où les calculatrices n'étaient pas aussi légères et bon marché qu'aujourd'hui. Elles contenaient des couples $\langle x, \log_{10}(x) \rangle$ pour un grand nombre de valeurs de x, qui pouvaient être utilisées pour faire des calculs complexes, tels l'extraction

de racines. Ainsi, le calcul de $\sqrt[9]{876}$, qui est long et difficile à faire à la main, revient à chercher dans la table la valeur de $\log_{10}(8.76) \approx 0.942504106$, calculer la fraction $\frac{0.942504106+2}{9} = 0.326944901$, et regarder dans la table l'antécédent de 0.326944901 par $\log_{10}$ — toutes opérations très simples.

Cet exemple est représentatif du fonctionnement de la compilation de connaissances : la phase hors ligne consiste à calculer les logarithmes, qui sont difficiles à obtenir *et* potentiellement utiles pour plusieurs opérations en ligne. Chaque entrée peut ensuite être utilisée dans un grand nombre de cas (ainsi, la valeur de $\log_{10}(8.76)$ sert à calculer la racine de 876, 87.6, $8.76\ldots$), et dans chacun de ces cas, la part de travail revenant à l'utilisateur est minime.

Prenons un autre exemple, plus proche du sujet : dans l'introduction, la première solution que nous proposions pour améliorer la réactivité du système était de résoudre le problème au préalable, pour toutes les situations possibles, de manière à fournir au programme de contrôle une table de règles de décision. Bien que naïve, cette idée relève également de la compilation de connaissances ; les trois aspects principaux de la définition sont en effet présents, à savoir que la phase hors ligne est chargée des calculs difficiles (la résolution du problème pour des situations initiales multiples) et souvent répétés (une même situation peut être rencontrée de nombreuses fois), et la phase en ligne est simple (regarder dans la table la prochaine décision à prendre).

Ce que désigne usuellement (et historiquement) le mot *compilation* est la traduction d'un programme depuis un langage de haut niveau, facilement compréhensible et modifiable par des êtres humains, vers du « langage machine », de bas niveau, beaucoup plus efficace d'un point de vue algorithmique. Il s'agit là encore de pré-traitement — on transforme quelque chose hors ligne pour faciliter son utilisation en ligne. Ce n'est cependant pas à proprement parler de la *compilation de connaissances*, domaine spécifique qui diffère de l'étude générale du pré-traitement sur deux points.

Premièrement, comme le souligne Marquis [Mar08], le nom de la « compilation de connaissances » limite sa portée aux « bases de connaissances », c'est-à-dire à des ensembles d'informations généralement représentées par des formules logiques, et à leur exploitation, c'est-à-dire au raisonnement automatisé. La définition reste très générale — le terme « information » peut recouvrir un grand nombre de choses — mais exclut notamment la compilation de programmes au sens classique.

La seconde différence entre compilation standard et compilation de connaissances est une conséquence de la précédente [CD97]. Étant liée à la représentation des connaissances et au raisonnement, la compilation de connaissances ne considère la difficulté d'un problème que par le prisme de la *théorie de la complexité algorithmique* [voir Pap94 pour un survol de la théorie de la complexité]. Ainsi, les auteurs travaillant sur la compilation ne cherchent pas simplement à rendre les problèmes *plus faciles*, mais *drastiquement* plus faciles, en les faisant changer de classe de complexité, par exemple de NP-complet à P, de PSPACE-complet à NP, etc.

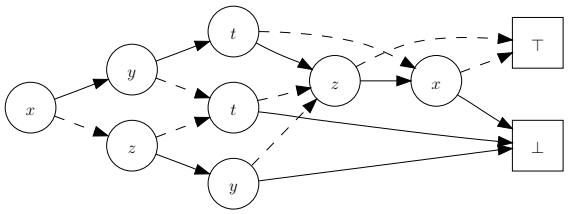


Fig. 1.1 : Un exemple de BDD. Les arcs continus (resp. pointillés) sont ceux étiquetés $\top$ (resp. $\bot$).

Dans le chapitre 2, nous montrons en quelle mesure résoudre un problème de planification revient à appliquer certaines opérations à certaines fonctions. Comme notre application, la conduite de systèmes autonomes, requiert une complexité en ligne minimale, nous nous sommes concentrés sur l'obtention d'algorithmes *polynomiaux* pour les opérations à faire en ligne. Cette hypothèse écarte de notre étude les représentations sur lesquelles le raisonnement en temps polynomial n'est pas possible [e.g. les logiques de haut niveau listées dans GK$^+$95]. Nous nous limitons en pratique aux structures de graphe, et au mécanisme d'inférence classique — nous laissons de côté les bases stratifiées ou pondérées, bien que leur compilation ait été étudiée [BYD07, DM04].

1.1.2 Exemple de langage-cible : OBDDs

Compiler un problème revient à en modifier la forme (cette modification étant potentiellement complexe), de façon à ce que le problème dans sa forme compilée soit soluble (c'est-à-dire polynomial en la taille de la forme compilée) tout en restant aussi compact que possible. Cela peut être vu comme une *traduction* depuis un *langage d'entrée* vers un *langage-cible*. Nous définirons formellement la notion de langage dans la section suivante [§ 1.2] ; avant cela, présentons de quoi il s'agit par le biais du langage bien connu des *diagrammes de décision binaires*.

Diagrammes de décision binaires

Introduits par Lee [Lee59] et Akers [Ake78], les *diagrammes de décision binaires (BDDs, binary decision diagrams)* sont des graphes acycliques orientés représentant des fonctions booléennes de variables booléennes. Ils ont exactement deux feuilles, étiquetées respectivement $\top$ (« vrai ») et $\bot$ (« faux ») ; leurs nœuds internes sont étiquetés par une variable booléenne et ont exactement deux arcs sortants, étiquetés respectivement $\top$ et $\bot$. Un exemple de BDD est présenté figure 1.1.

Un BDD représente une fonction, au sens où il associe une unique valeur booléenne à toute *instanciation* des variables qu'il mentionne. Nous illustrons ceci sur l'exemple de la figure 1.1. Quatre variables sont mentionnées ; chacune peut prendre deux valeurs. En choisissant une valeur pour chaque, par exemple $x = \top$, $y = \bot$, $z = \top$ et $t = \bot$, on obtient une instanciation de ces quatre variables parmi

les $2^4 = 16$ possibles. Comment le BDD associe-t-il une valeur de vérité à cette instanciation ? Il suffit de partir de la racine et de suivre un chemin jusqu'à l'une des feuilles. Le chemin est complètement déterminé par l'instanciation choisie : chaque nœud correspond à une variable, l'arc suivant étant celui étiqueté par la valeur assignée à cette variable dans l'instanciation. Le chemin mène soit à la feuille $\top$, soit à la feuille $\bot$: l'étiquette est la valeur qu'associe le BDD à l'instanciation choisie.

Le nom de « diagramme de décision binaire » retranscrit ce comportement : en partant de la racine, chaque nœud correspond à une décision possible — « quelle valeur choisir pour cette variable ? » — qui dirige vers le sous-graphe associé — « pour telle décision, aller par là, sinon par là ». Notons que chaque instanciation possible correspond à exactement un chemin ; les BDDs représentent donc bien des *fonctions* (et non des *relations*).

La famille des diagrammes de décision

En imposant des restrictions structurelles aux BDDs, on obtient d'intéressants *sous-langages*. Par exemple, un *free BDD (FBDD)* [GM94] est un BDD satisfaisant la propriété de lecture unique (*read-once*) : chaque chemin ne peut contenir au plus qu'une occurrence de chaque variable. Mais le sous-langage de BDD le plus influent est sans conteste celui des *BDDs ordonnés (OBDDs, ordered BDDs)* [Bry86], dans lesquels l'ordre des variables doit être le même le long de tous les chemins. La figure 1.2 montre un FBDD et un OBDD représentant la même fonction que le BDD de la figure 1.1.

Le concept de « diagramme de décision » n'est pas intrinsèquement lié à l'algèbre booléenne ; l'idée en a été étendue aux variables non-booléennes, donnant les *diagrammes de décision multivalués (MDDs, multi-valued decision diagrams)* [voir section 1.3.6], ainsi qu'aux fonctions non-booléennes, donnant les *diagrammes de décision algébriques (ADDs, algebraic decision diagrams)* [BF$^+$97]. La figure 1.3 donne un exemple de MDD.

Les OBDDs sont compacts et efficaces

Nous avons expliqué ce que sont les OBDDs, et leur fonctionnement ; mais cela ne suffit pas à montrer en quoi ils constituent un bon langage-cible de compilation. Informellement, il y a deux raisons à cela : ils sont efficaces, et ils sont compacts. Ces aspects sont bien sûr indissociables. Ainsi, il existe des façons plus compactes que les OBDDs de représenter des fonctions booléennes, comme les formules propositionnelles. Les OBDDs ne sont pas non plus les structures les plus efficaces pour toute application — la conjonction de plusieurs fonctions est plus facile à construire en utilisant des tables de vérité, par exemple. Ce qui rend les OBDDs avantageux, c'est le fait qu'ils soient *à la fois* efficaces et compacts — ils constituent un bon compromis entre l'efficacité et la taille.

Leur compacité tient à leur structure de graphe, qui permet de *factoriser* les sous-graphes « identiques », c'est-à-dire ceux qui représentent la même fonction — ils sont dits *isomorphes*. Cette factorisation est effectuée par une opération de

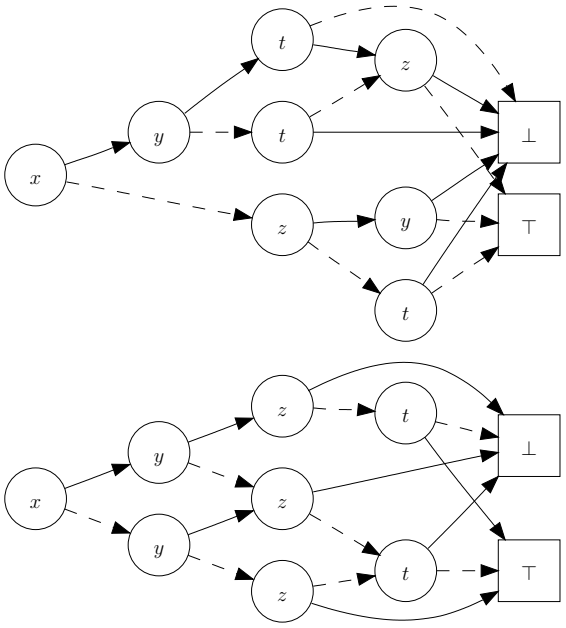


Fig. 1.2 : Un FBDD (en haut) et un OBDD (en bas), représentant tous deux la même fonction que le BDD de la figure 1.1.

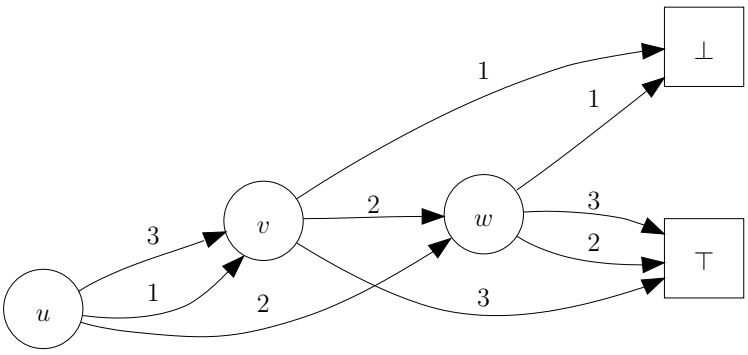


Fig. 1.3 : Un exemple de MDD, sur les variables u, v et w de domaine $\{1, 2, 3\}$.

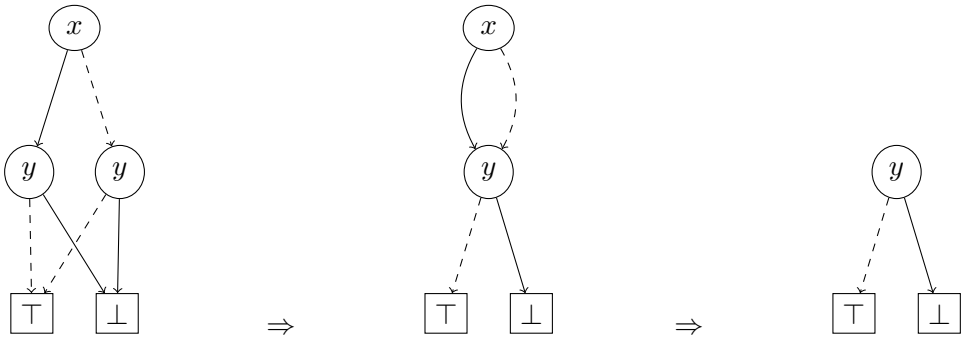


Fig. 1.4 : Illustration de la procédure de réduction des diagrammes de décision binaires. Le BDD de gauche représente la formule $(x \wedge \neg y) \vee (\neg x \wedge \neg y)$. La réduction fusionne les deux nœuds y, qui sont *isomorphes* (ils représentent la même fonction) ; le résultat est le BDD du milieu. Le nœud x est alors retiré, car il est *redondant* : la valeur de x n'a pas d'importance. Le BDD réduit est à droite.

réduction ayant une complexité polynomiale, décrite par Bryant [Bry86] et illustrée figure 1.4. Non seulement cette procédure permet un gain exponentiel en espace dans certains cas, mais elle est également la clé de l'efficacité des OBDDs. Le fait qu'elle soit polynomiale est important, car cela permet de ne considérer que des OBDDs réduits (nous le ferons toujours, implicitement, dans la suite).

On évalue l'efficacité des OBDDs au regard de leurs performances pour des opérations courantes. Vérifier s'il existe une instanciation satisfaisant une formule propositionnelle est difficile, mais peut cependant être fait en temps constant si la formule est représentée par un OBDD (en effet, les OBDDs n'étant satisfaits par aucune instanciation sont ceux qui se réduisent à une simple feuille $\bot$). Mieux encore, vérifier si deux OBDDs utilisant le même ordre de variables représentent exactement la même fonction peut être effectué en temps linéaire en la somme de leurs tailles respectives. Qui plus est, construire un OBDD représentant la conjonction (ou la disjonction, ou encore l'application de n'importe quel opérateur booléen) de deux OBDDs de même ordre est seulement linéaire en le produit de leurs tailles respectives. Ces remarquables propriétés furent découvertes par Bryant [Bry86].

Les OBDDs ne sont pas le seul langage à être aussi efficaces : ainsi, toutes les opérations que nous avons mentionnées ont une complexité au pire cas linéaire sur les tables de vérité. Cependant, les OBDDs possèdent toutes ces propriétés tout en étant potentiellement exponentiellement plus compacts que les tables de vérité. En effet, la quantité de mémoire nécessaire à stocker une table de vérité est proportionnelle au nombre d'instanciations satisfaisant la formule représentée, alors que ce n'est pas le cas pour les OBDDs.

OBDDs : le meilleur langage ?

Comme ils constituent un si bon compromis entre efficacités spatiale et opérationnelle, les OBDDs sont très largement utilisés depuis des années, et ce pour de

multiples applications. Doit-on en déduire qu'il s'agit là de la meilleure façon de compiler des fonctions booléennes ? La réponse générale est *non*, car cela dépend de l'application ciblée. Quand il est souvent nécessaire de vérifier l'équivalence de deux fonctions ou d'en calculer la conjonction, les OBDDs sont probablement le meilleur choix ; mais si le test d'équivalence est superflu, et que seuls des tests de satisfaction (*model-checking*) sont nécessaires, alors les BDDs — moins contraints — sont bien plus adaptés, puisqu'ils peuvent consommer exponentiellement moins d'espace.

*
**

Choisir un language-cible pour une application donnée est donc une étape de conception importante, qui ne doit pas être négligée, car un mauvais choix peut conduire à de grandes pertes de performances en temps ou en espace — ceci étant particulièrement crucial dans le cas de systèmes embarqués. Ce choix est compliqué par l'existence de nombreux langages-cibles ; des outils de comparaison de langages sont donc nécessaires. Introduite par Darwiche et Marquis [DM02], la *carte de compilation* fournit de tels outils ; nous les présentons dans la section suivante.

1.2 Un cadre pour la cartographie des langages

Le but de la *carte de compilation* est de comparer les langages selon leur complexité spatiale et leur efficacité (en termes de complexité au pire cas) pour diverses opérations possibles, comme la requête d'informations sur la forme compilée, ou l'application de transformations. La carte facilite la comparaison des fragments sur le plan de la complexité théorique, permettant ainsi de ne retenir que les meilleurs pour chaque application.

L'objectif de cette section est de présenter formellement la carte de compilation, notamment la notion de *langage* sur laquelle elle repose, et les concepts en permettant la comparaison.

1.2.1 Notations

Avant d'introduire nos conventions de notation, précisons que cette thèse inclut un index des symboles [p. 173].

On note $\mathbb{R}$ l'ensemble des nombres réels, $\mathbb{Z}$ celui des entiers relatifs, $\mathbb{N}$ celui des entiers naturels (y compris 0), $\mathbb{N}^*$ celui des entiers naturels non nuls, et $\mathbb{B}$ celui des constantes booléennes. Ces dernières sont notées $\top$ et $\bot$ (respectivement pour « vrai » et « faux »), mais sont souvent identifiées aux entiers 1 et 0, de sorte que :

$$\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}.$$

On utilise $\mathbb{S}A$ pour l'ensemble des singletons d'un ensemble A. Ainsi, par exemple, $\mathbb{SB} = \big\{\{\top\}, \{\bot\}\big\}$, et $\mathbb{SN} = \bigcup_{n\in\mathbb{N}} \big\{\{n\}\big\}$.

Dans une optique de généralisation, nous considérerons des variables de domaine quelconque — discret ou continu, fini ou infini… mais non vide. On note $\mathcal{V}$ l'ensemble de toutes les variables possibles ; $\mathcal{V}$ est bien sûr non vide, et on le suppose ordonné par un ordre strict total $<_{\mathcal{V}}$. Quand on considérera des ensembles de variables indicées, comme $\{x_1, x_2 \ldots, x_k\}$, on supposera toujours implicitement que $x_1 <_{\mathcal{V}} x_2 <_{\mathcal{V}} \ldots <_{\mathcal{V}} x_k$.

Pour $x \in \mathcal{V}$, on note $\mathrm{Dom}(x)$ son *domaine*, qui peut être n'importe quel ensemble non vide. Il est néanmoins souvent nécessaire de considérer des variables définies sur des domaines spécifiques : pour S un ensemble, on définit $\mathcal{V}_S = \{\, x \in \mathcal{V} \mid \mathrm{Dom}(x) = S \,\}$. Parmi les ensembles notables de variables, citons $\mathcal{B}$, l'ensemble des variables booléennes, défini simplement comme $\mathcal{V}_{\mathbb{B}}$, et $\mathcal{E}$, l'ensemble des variables énumérées, que nous définissons arbitrairement comme ayant pour domaine un intervalle fini d'entiers (ainsi $\mathcal{E} = \bigcup_{(a,n)\in\mathbb{Z}\times\mathbb{N}} \mathcal{V}_{\{a,a+1,\ldots,a+n\}}$).

Pour $X = \{x_1, \ldots, x_k\} \subseteq \mathcal{V}$, on note $\mathrm{Dom}(X)$ l'ensemble des *instanciations des variables de* X (ou X*-instanciations*), c'est-à-dire que $\mathrm{Dom}(X) = \mathrm{Dom}(x_1) \times \mathrm{Dom}(x_2) \times \cdots \times \mathrm{Dom}(x_k)$. La notation $\vec{x}$ désigne une X-instanciation, i.e., $\vec{x} \in \mathrm{Dom}(X)$. L'ensemble X est appelé *support* de $\vec{x}$. L'ensemble vide n'a qu'une seule instanciation, notée $\vec{\varnothing}$.

Soient $X, Y \subseteq \mathcal{V}$, et soit $\vec{x}$ une X-instanciation. La *restriction* de $\vec{x}$ aux variables de Y, notée $\vec{x}_{|Y}$, est la $X \cap Y$-instanciation dans laquelle chaque variable prend la même valeur que dans $\vec{x}$. À noter que cette définition autorise Y à être disjoint de X (auquel cas $\vec{x}_{|Y}$ est toujours égal à $\vec{\varnothing}$). Pour une variable $x_i \in X$ donnée, $\vec{x}_{|\{x_i\}}$ fournit donc la valeur prise par x_i dans $\vec{x}$; on l'écrit plus simplement $\vec{x}_{|x_i}$.

Soient $X, Y \subseteq \mathcal{V}$, et soient $\vec{x}$ et $\vec{y}$ des instanciations des variables de X et Y. Si X et Y sont disjoints, $\vec{x} \,.\, \vec{y}$ est la *concaténation* des deux instanciations, c'est-à-dire la $X \cup Y$-instanciation qui coïncide avec $\vec{x}$ pour les variables de X et avec $\vec{y}$ pour les variables de Y.

Étant donné deux ensembles S et E, on utilise la notation E^S pour désigner l'ensemble de toutes les fonctions de la forme $f : S \to E$; S est appelé l'*ensemble de départ* de f, et E son *ensemble de valuation*. La restriction de f à un ensemble S', que l'on note $f_{|S'}$, est la fonction de $S \cap S'$ dans E qui coïncide avec f sur $S \cap S'$. On considérera généralement des fonctions pour lesquelles $S = \mathrm{Dom}(V)$, avec $V \subseteq \mathcal{V}$ un ensemble de variables ; on les appelle fonctions des variables de V dans E. Pour une telle fonction $f : \mathrm{Dom}(V) \to E$, V est appelé la *portée* de f, et est noté $\mathrm{Scope}(f)$. Précisons que les fonctions peuvent prendre en entrée des instanciations dont le support est plus grand que leur portée : on considère toujours implicitement que $f(\vec{v})$ signifie $f(\vec{v}_{|\,\mathrm{Scope}(f)})$.

1.2.2 Langage de représentation

Nous allons maintenant présenter les éléments de base de la carte : appelés *langages de représentation*, ils furent introduits dans le cadre propositionnel par

Fargier et Marquis [FM09]. Nous étendons leur définition pour englober également les variables non propositionnelles.

Ce qui est exprimé : le domaine d'interprétation

De manière générale, les langages-cibles de compilation représentent des *fonctions*. Elles sont de différentes sortes (booléennes, réelles, à variables booléennes, énumérées, etc.) ; la catégorie de fonctions que représente un langage constitue son *domaine d'interprétation*.

Définition 1.2.1 (Domaine d'interprétation)**.** Soit $V \subseteq \mathcal{V}$ un ensemble *fini* de variables, et E un ensemble quelconque. Le *domaine d'interprétation* associé à V et E est l'ensemble

$$\mathfrak{D}_{V,E} = \bigcup_{V' \subseteq V} E^{\mathrm{Dom}(V')}$$

des fonctions portant sur des variables de V et renvoyant des éléments de E.

Ainsi, $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$ est l'ensemble des fonctions booléennes à variables booléennes ; c'est le domaine d'interprétation des OBDDs [§ 1.1.2].

Comment l'exprimer : les structures de données

La compilation de connaissances vise à exprimer les fonctions comme des instances de *structures de données* spécifiques. Une structure de données [voir e.g. CL$^+$01, partie III] est une organisation particulière des informations dans la mémoire d'un ordinateur, munie d'algorithmes de manipulation exploitant les données efficacement, en tirant parti de cette organisation. Parmi les structures de données bien connues, on peut citer les piles, les files, les tables de hachage, les arbres, les graphes…

Un objet mathématique comme une fonction est un concept abstrait. Pour le représenter avec une structure de données, il est nécessaire de le *modéliser*, en identifiant les opérations de base par le biais desquelles il est manipulé ; ainsi, les *ensembles* sont des « choses » qu'il est possible d'*intersecter*, d'*unir*, d'*énumérer*, etc. Une fois qu'un modèle est décrit, il est possible de définir une structure de donnée l'implantant efficacement. Celle-ci n'est pas nécessairement unique ; diverses structures données peuvent être utilisées pour exprimer un même concept mathématique. Leur efficacité peut varier, mais aucune n'est plus « correcte » qu'une autre.

Nous considérons les structures de données d'une manière relativement abstraite, sans nous apesantir sur les détails d'implantation. Nous utiliserons un attribut important des structures de données, la *taille mémoire* de leurs instances. Pour cela, on munit chaque structure de données d'une *fonction de taille caractéristique* abstraite, associant à chaque instance φ de la structure de données en question, un entier positif représentatif de la place qu'elle prend en mémoire. La taille caractéristique d'une instance φ, que nous notons $\|\varphi\|$ pour la distinguer du cardinal d'un ensemble (noté $|S|$), n'est pas directement égale à sa taille mémoire $\mathrm{Mem}(\varphi)$. Le point-clef est que la taille caractéristique doit croître de la même manière que la taille mémoire :

$\|\varphi\| \in \Theta(\text{Mem}(\varphi))$, c'est-à-dire qu'il existe deux constantes positives k_1 et k_2 telles que lorsque φ est suffisamment gros, $\text{Mem}(\varphi) \cdot k_1 \leqslant \|\varphi\| \leqslant \text{Mem}(\varphi) \cdot k_2$.

Par exemple, considérons une structure de données destinée à représenter des listes de nombres réels, et une autre destinée à représenter des listes de nombres entiers. Il est probable que les instances de la première prennent plus de place en mémoire que les instances de la seconde, étant donné que les nombres réels sont plus coûteux que les entiers en termes de mémoire. Pourtant, si on utilise dans les deux cas des structures exactement similaires excepté le type utilisé pour les nombres, elles ont la même *taille caractéristique*, puisqu'elles sont équivalentes modulo une simple constante multiplicative.

Une instance particulière d'une structure de données particulière peut représenter divers objets mathématiques. Ainsi, la liste chaînée contenant $\boxed{\text{a}} \rightarrow \boxed{\text{b}} \rightarrow \boxed{\text{c}}$ peut exprimer le tuple $\langle a, b, c\rangle$, l'ensemble $\{a, b, c\}$, la fonction de $\{1, 2, 3\}$ vers $\{a, b, c\}$ associant a à 1, b à 2 et c à 3, etc. Cette interprétation dépend exclusivement du contexte dans lequel elle est utilisée : c'est là qu'intervient la notion de *langage*.

Relier domaine d'interprétation et structure de données : les langages

Nous pouvons à présent introduire les langages de représentation, en suivant Fargier et Marquis [FM09].

Définition 1.2.2. Un *langage de représentation* est un triplet $\mathtt{L} = \langle \mathfrak{D}_{V,E}, \mathcal{R}, [\![\cdot]\!]\rangle$, où :

- $\mathfrak{D}_{V,E}$ est un domaine d'interprétation, sur un ensemble fini de variables $V \subseteq \mathcal{V}$ (appelé *portée* de $\mathtt{L}$) et un ensemble E (appelé *ensemble de valuation* de $\mathtt{L}$) ;
- $\mathcal{R}$ est un ensemble d'instances d'une certaine structure de données, qui sont appelées $\mathtt{L}$-*représentations* ;
- $[\![\cdot]\!] : \mathcal{R} \rightarrow \mathfrak{D}_{V,E}$ est la *fonction d'interprétation* (ou *sémantique*) de $\mathtt{L}$, qui à *toute* $\mathtt{L}$-représentation associe une fonction portant sur des variables de V et renvoyant des éléments de E.

Un langage de représentation est donc un simple ensemble de structures, les *représentations*, chacune étant associée à une certaine fonction, que l'on nomme son *interprétation*. Cette définition très générale[1] nous permet d'introduire plusieurs notions (sous-langage, fragment, compacité…) sans faire d'hypothèse sur les variables ou les structures de données concernées.

Pour simplifier les définitions, un langage de représentation noté $\mathtt{L}$ sera toujours implicitement donné par $\mathtt{L} = \langle \mathfrak{D}_{V_\mathtt{L},E_\mathtt{L}}, \mathcal{R}_\mathtt{L}, [\![\cdot]\!]_\mathtt{L}\rangle$. Pour une $\mathtt{L}$-représentation φ, $\text{Scope}_\mathtt{L}(\varphi)$ désigne l'ensemble des variables sur lesquelles porte l'interprétation de φ : $\text{Scope}_\mathtt{L}(\varphi) = \text{Scope}([\![\varphi]\!]_\mathtt{L})$. En l'absence d'ambiguïté, on omettra l'indice $\mathtt{L}$, écrivant alors simplement $[\![\varphi]\!]$ et $\text{Scope}(\varphi)$.

[1] Elle est notamment plus générale que la définition originelle [FM09], qui se limite en particulier au domaine d'interprétation $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$.

Pour illustrer cette notion de langage de représentation, prenons l'exemple des BDDs. La section 1.1.2 a présenté la structure de données « diagramme de décision binaire », et expliqué comment on pouvait l'interpréter comme une fonction booléenne sur des variables booléennes. Nous avons là tous les éléments d'un langage de représentation : si on note $\mathcal{R}_{\text{BDD}}$ l'ensemble de tous les BDDs, et $[\![\cdot]\!]_{\text{BDD}}$ la fonction d'interprétation que nous avons décrite informellement, alors on peut définir le langage de représentation des BDDs par $\mathtt{L}_{\text{BDD}} = \langle \mathfrak{D}_{\mathcal{B},\mathbb{B}}, \mathcal{R}_{\text{BDD}}, [\![\cdot]\!]_{\text{BDD}} \rangle$.

Les langages de représentation forment une *hiérarchie*, en ce que chaque langage est obtenu en imposant une propriété restrictive à un langage plus général ; comme illustré dans la section 1.1.2, les FBDDs sont des BDDs *read-once*, les OBDDs des FBDDs ordonnés, et ainsi de suite. Cette idée nous mène à la définition d'un sous-langage.

Définition 1.2.3 (Sous-langage)**.** Soit $\mathtt{L}_1$ et $\mathtt{L}_2$ deux langages de représentation. $\mathtt{L}_2$ est un *sous-langage* de $\mathtt{L}_1$ (ce que l'on note $\mathtt{L}_2 \subseteq \mathtt{L}_1$) si et seulement si chacune des propriétés suivantes est vérifiée :

- $\mathfrak{D}_{V_{\mathtt{L}_2},E_{\mathtt{L}_2}} \subseteq \mathfrak{D}_{V_{\mathtt{L}_1},E_{\mathtt{L}_1}}$;
- $\mathcal{R}_{\mathtt{L}_2} \subseteq \mathcal{R}_{\mathtt{L}_1}$;
- $[\![\cdot]\!]_{\mathtt{L}_2} = ([\![\cdot]\!]_{\mathtt{L}_1})_{|\mathcal{R}_{\mathtt{L}_2}}$.

Il est donc possible d'obtenir un sous-langage en restreignant les variables, les valeurs, ou les représentations d'un langage plus général ; mais l'interprétation des représentations du langage fils doit rester la même que dans le langage père. En appliquant cette définition, on peut considérer le langage des OBDDs comme un sous-langage de celui des MDDs [voir section 1.1.2], $\mathtt{L}_{\text{OBDD}} \subseteq \mathtt{L}_{\text{MDD}}$, puisque ces deux langages ont les mêmes propriétés structurelles et le même domaine de valuation (booléen), mais sont définis sur des variables différentes — les variables entières étant plus générales que les booléennes.

Notons qu'il est généralement impossible de restreindre le domaine d'interprétation d'un langage sans lui retirer des représentations, puisque la fonction d'interprétation $[\![\cdot]\!]$ doit rester la même *et* être définie sur toutes les représentations du sous-langage. On peut le constater grâce à la définition suivante.

Définition 1.2.4 (Restriction des variables)**.** Soit $\mathtt{L}$ un langage de représentation, et $X \subseteq \mathcal{V}$. La *restriction de* $\mathtt{L}$ *aux variables de* X, notée $\mathtt{L}_X$, est le sous-langage $\mathtt{L}' \subseteq \mathtt{L}$ le plus général (relativement à l'inclusion des ensembles de représentations) qui vérifie $E_{\mathtt{L}'} = E_{\mathtt{L}}$ et $V_{\mathtt{L}'} = X \cap V_{\mathtt{L}}$.

Nous utiliserons principalement cette définition pour restreindre les langages à des variables booléennes ; on utilise alors la simple notation $\mathtt{L}_{\mathcal{B}}$ pour désigner la restriction de $\mathtt{L}$ aux variables booléennes. Ainsi $\mathtt{L}_{\text{OBDD}} = (\mathtt{L}_{\text{MDD}})_{\mathcal{B}}$. Comme précisé plus haut, restreindre le domaine d'interprétation de $\mathtt{L}_{\text{MDD}}$ suffit à retirer un certain nombre de représentations : toutes celles qui *mentionnent* une variable non-booléenne ne peuvent être des $(\mathtt{L}_{\text{MDD}})_{\mathcal{B}}$-représentations, puisqu'elles seraient alors

interprétées comme des fonctions sur des variables non-booléennes. La proposition suivante formalise cette idée.

Proposition 1.2.5. Soit $\mathtt{L}$ un langage de représentation, et $X \subseteq \mathcal{V}$. L'ensemble des représentations de $\mathtt{L}_X$ est $\{ \varphi \in \mathcal{R}_\mathtt{L} \mid \text{Scope}_\mathtt{L}(\varphi) \subseteq X \}$.

Grâce à ceci, il est d'ores et déjà possible de donner une propriété simple, mais utile sur les sous-langages : la restriction des variables maintient la hiérarchie des langages.

Proposition 1.2.6. Si $\mathtt{L}$ et $\mathtt{L}'$ sont deux langages de représentation, et $X \subseteq \mathcal{V}$ un ensemble de variables, alors

$$\mathtt{L} \subseteq \mathtt{L}' \quad \Longrightarrow \quad \mathtt{L}_X \subseteq \mathtt{L}'_X.$$

En résumé, notre définition de sous-langage permet d'identifier comme faisant partie de la même « famille », des langages portant sur des domaines d'interprétation différents. Cependant, dans la hiérarchie des langages, le passage d'un langage père à un sous-langage est le plus souvent le fait d'une restriction *structurelle*. Ainsi, les OBDDs sont des BDDs particuliers, dans lesquels les variables sont rencontrées dans le même ordre quel que soit le chemin ; mais ces deux langages portent sur les mêmes variables et le même domaine de valuation. Pour refléter cette nuance, on raffine la notion de sous-langage.

Définition 1.2.7 (Fragment)**.** Soit $\mathtt{L}$ un langage de représentation. Un *fragment* de $\mathtt{L}$ est un sous-langage $\mathtt{L}' \subseteq \mathtt{L}$ tel que $V_{\mathtt{L}'} = V_\mathtt{L}$ et $E_{\mathtt{L}'} = E_\mathtt{L}$.

Un fragment est donc un sous-langage particulier, qui a le même domaine d'interprétation que son parent. $\mathtt{L}_{\text{OBDD}}$ est un fragment de $\mathtt{L}_{\text{BDD}}$, mais pas un fragment de $\mathtt{L}_{\text{MDD}}$. Dans la littérature, il n'existe, à notre connaissance, pas de distinction entre sous-langage et fragment ; nous l'introduisons ici pour clarifier les définitions. La notation suivante a également cet objectif.

Définition 1.2.8 (Opérations sur des fragments)**.** Soit $\mathtt{L}$ un langage de représentation, et $\mathcal{P}$ une propriété sur les $\mathtt{L}$-représentations (c'est-à-dire une fonction associant une valeur booléenne à tout élément de $\mathcal{R}_\mathtt{L}$).

La *restriction de* $\mathtt{L}$ *à* $\mathcal{P}$ (aussi appelé *fragment de* $\mathtt{L}$ *satisfaisant* $\mathcal{P}$) est le fragment $\mathtt{L}' \subseteq \mathtt{L}$ vérifiant

$$\forall \varphi \in \mathcal{R}_\mathtt{L}, \qquad \mathcal{P}(\varphi) \iff \varphi \in \mathcal{R}_{\mathtt{L}'}.$$

Soit $\mathtt{L}_1$ et $\mathtt{L}_2$ deux fragments de $\mathtt{L}$. L'*intersection* (resp. l'*union*) de $\mathtt{L}_1$ et $\mathtt{L}_2$ est le fragment de $\mathtt{L}$ dont l'ensemble des représentations est exactement l'intersection (resp. l'union) des ensembles de représentations de $\mathtt{L}_1$ et $\mathtt{L}_2$.

1.2.3 Comparaison de langages

L'évaluation de l'efficacité d'un langage pour une application donnée se base sur quatre critères généraux :

1. l'expressivité (quelles fonctions est-il capable de représenter ?) ;
2. la compacité (combien de place cela prend-il de représenter des fonctions ?) ;
3. l'efficacité des requêtes (combien de temps cela prend-il d'obtenir des informations sur les fonctions ?) ;
4. l'efficacité des transformations (combien de temps cela prend-il d'appliquer des opérations sur les fonctions ?).

Les deux premiers critères visent à comparer les langages sur le plan de l'efficacité spatiale, tandis que les deux derniers permettent de les comparer sur le plan de l'efficacité temporelle. Dans la suite, on définit formellement ces notions sur les langages de représentation en général ; dans la pratique, ils sont principalement utilisés pour comparer des *fragments*, voire des *sous-langages* d'un langage donné.

Efficacité spatiale

La notion d'*expressivité* a été introduite, sous la forme d'un préordre sur les langages, par Gogic et al. [GK$^+$95].

Définition 1.2.9 (Expressivité relative)**.** Soit $\mathtt{L}_1$ et $\mathtt{L}_2$ deux langages de représentation. $\mathtt{L}_1$ est *au moins aussi expressif que* $\mathtt{L}_2$, ce que l'on note $\mathtt{L}_1 \leqslant_e \mathtt{L}_2$, si et seulement si pour toute $\mathtt{L}_2$-représentation φ_2, il existe une $\mathtt{L}_1$-représentation φ_1 telle que $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

La relation $\leqslant_e$ est clairement réflexive et transitive ; il s'agit donc d'un préordre (partiel), dont on note $\sim_e$ la partie symétrique et $<_e$ la partie asymétrique[2].

Deux langages de portées ou d'ensembles de valuation disjoints sont évidemment incomparables selon $\leqslant_e$. C'est la raison pour laquelle ce critère est un peu à part : l'utilisateur est censé connaître l'expressivité dont il a besoin pour son application, il va donc simplement écarter les langages ne lui permettant pas de représenter les fonctions voulues. Mais même lorsque deux langages sont comparables selon $\leqslant_e$, il est peu probable que leur expressivité *relative* soit très pertinente pour l'utilisateur. Par exemple, savoir que le langage des formules de Horn est strictement moins expressif que le langage des CNFs ne l'aide pas vraiment à choisir ; il est en revanche fondamental de savoir que le premier est un langage *incomplet*, contrairement au second.

Pour définir la complétude d'un langage, nous introduisons tout d'abord son expressivité (absolue).

[2] Soit $\preceq$ un préordre partiel sur un ensemble S. On définit la partie symétrique $\sim$ et la partie asymétrique $\prec$ de $\preceq$ par

$$\forall \langle a, b\rangle \in S^2, \quad a \sim b \iff a \preceq b \wedge a \succeq b,$$
$$\forall \langle a, b\rangle \in S^2, \quad a \prec b \iff a \preceq b \wedge a \not\succeq b.$$

Définition 1.2.10 (Expressivité)**.** L'*expressivité* d'un langage de représentation $\mathtt{L}$ est l'ensemble $\mathrm{Expr}(\mathtt{L}) = \{ f \in \mathfrak{D}_{V_\mathtt{L},E_\mathtt{L}} \mid \exists \varphi \in \mathcal{R}_\mathtt{L}, [\![\varphi]\!]_\mathtt{L} = f \}$ (on peut le voir comme l'image de $\mathcal{R}_\mathtt{L}$ par $[\![\cdot]\!]_\mathtt{L}$).

On définit simplement l'expressivité d'un langage comme l'ensemble des fonctions qu'il permet de représenter. Notons qu'il vient en particulier que $\mathtt{L}_2 \geqslant_e \mathtt{L}_1 \iff \mathrm{Expr}(\mathtt{L}_2) \subseteq \mathrm{Expr}(\mathtt{L}_1)$. Nous pouvons à présent introduire la complétude.

Définition 1.2.11 (Langage complet)**.** Un langage de représentation $\mathtt{L}$ est *complet* si et seulement si $\mathrm{Expr}(\mathtt{L}) = \mathfrak{D}_{V_\mathtt{L},E_\mathtt{L}}$.

Pour être complet, un langage doit être capable de représenter n'importe quelle fonction *de son domaine d'interprétation*. Par exemple, le langage susmentionné des BDDs, $\mathtt{L}_{\mathrm{BDD}}$, est complet ; cependant le langage des « BDDs ayant au plus 3 nœuds », défini sur le même domaine d'interprétation, est incomplet (il est en effet impossible de représenter la formule propositionnelle $x \vee y \vee z \vee t$ par un BDD n'ayant que 3 nœuds). La complétude est définie en fonction du domaine d'interprétation ; deux langages qui ne diffèrent que par leur domaine d'interprétation ont la même expressivité, mais pas nécessairement la même complétude. Ainsi, le langage des BDDs défini sur le domaine d'interprétation $\mathfrak{D}_{\mathcal{V}_{\mathbb{R},\mathbb{B}}}$ est incomplet. Au final, un *fragment* d'un langage incomplet *ne peut pas* être complet, mais un *sous-langage* peut l'être. On rencontrera des exemples plus réalistes de langages incomplets dans la suite, comme le langage des clauses de Horn sur le domaine d'interprétation $\mathfrak{D}_{\mathcal{B},\mathbb{B}}$ [§ 1.3.3.3].

Une fois que l'utilisateur a identifié des langages assez expressifs pour son application, il doit pouvoir comparer leur efficacité en termes d'espace mémoire. C'est le rôle de la relation de *compacité*, introduite elle aussi par Gogic et al. [GK+95] (et détaillée par la suite par Darwiche et Marquis [DM02]).

Définition 1.2.12 (Compacité)**.** Soit $\mathtt{L}_1$ et $\mathtt{L}_2$ deux langages de représentation. $\mathtt{L}_1$ est *au moins aussi compact* que $\mathtt{L}_2$, ce que l'on note $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$, si et seulement s'il existe un polynôme $P(\cdot)$ tel que pour toute $\mathtt{L}_2$-représentation φ_2, il existe une $\mathtt{L}_1$-représentation φ_1 vérifiant $\|\varphi_1\| \leqslant P(\|\varphi_2\|)$ et $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

De la même manière que $\leqslant_e$, $\leqslant_s$ est un préordre partiel, duquel on note $\sim_s$ la partie symétrique et $<_s$ la partie asymétrique. Le préordre $\leqslant_s$ est un raffinement de $\leqslant_e$, puisque pour n'importe quels langages de représentation $\mathtt{L}_1$ et $\mathtt{L}_2$, on a $\mathtt{L}_1 \leqslant_s \mathtt{L}_2 \implies \mathtt{L}_1 \leqslant_e \mathtt{L}_2$.

Il est important de remarquer que la compacité ne requiert que l'*existence* d'un équivalent de taille polynomiale, que cet équivalent soit calculable en temps polynomial ou non. Une condition suffisante pour que $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ est qu'il existe un algorithme *en espace polynomial* associant à chaque $\mathtt{L}_2$-représentation une $\mathtt{L}_1$-représentation de même interprétation. Si l'on impose à l'algorithme en question d'être de surcroît polynomial en temps (ce qui est plus restrictif — $\mathsf{P} \subsetneq \mathsf{PSPACE}$ sauf si $\mathsf{P} = \mathsf{NP}$ [voir Pap94]), on obtient la relation suivante, introduite par Fargier et Marquis [FM09].

Définition 1.2.13 (Traduisibilité polynomiale)**.** Soit $\mathtt{L}_1$ et $\mathtt{L}_2$ deux langages de représentation. $\mathtt{L}_2$ est *polynomialement traduisible* vers $\mathtt{L}_1$, ce que l'on note $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, si et seulement s'il existe un algorithme en temps polynomial associant à chaque $\mathtt{L}_2$-représentation φ_2, une $\mathtt{L}_1$-représentation φ_1 vérifiant $[\![\varphi_1]\!]_{\mathtt{L}_1} = [\![\varphi_2]\!]_{\mathtt{L}_2}$.

Une nouvelle fois, $\leqslant_p$ est un préordre partiel, dont on note $\sim_p$ la partie symétrique et $<_p$ la partie asymétrique ; c'est un raffinement de la compacité, puisque tous les langages de représentation $\mathtt{L}_1$ et $\mathtt{L}_2$ satisfont $\mathtt{L}_1 \leqslant_p \mathtt{L}_2 \implies \mathtt{L}_1 \leqslant_s \mathtt{L}_2$. Il est assez clair que si $\mathtt{L}_2$ est un sous-langage de $\mathtt{L}_1$, alors $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$ (chaque $\mathtt{L}_2$-représentation *est* une $\mathtt{L}_1$-représentation, l'algorithme est donc trivial).

La proposition suivante résume les relations entre les trois préordres que nous venons d'introduire.

Proposition 1.2.14. Si $\mathtt{L}_1$ et $\mathtt{L}_2$ sont des langages de représentation, alors

$$\mathtt{L}_2 \subseteq \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_p \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_s \mathtt{L}_1 \implies \mathtt{L}_2 \geqslant_e \mathtt{L}_1.$$

Ces relations de comparaison ont en commun le fait qu'elles sont maintenues par la restriction des deux langages à un même ensemble de variables.

Proposition 1.2.15. Soit $\mathtt{L}$ et $\mathtt{L}'$ deux langages de représentation, et $\leqslant$ l'un des trois préordres de comparaison que nous avons définis, $\leqslant_p$, $\leqslant_s$ ou $\leqslant_e$. Pour n'importe quel ensemble de variables $X \subseteq \mathcal{V}$, on a

$$\mathtt{L} \leqslant \mathtt{L}' \quad \implies \quad \mathtt{L}_X \leqslant \mathtt{L}'_X.$$

Nous avons à présent des outils permettant de comparer les langages en termes d'expressivité et d'efficacité spatiale, mais cela n'est pas suffisant pour permettre à un utilisateur de choisir un « bon » langage pour son application. En effet, il existe de nombreuses opérations qu'il peut vouloir appliquer sur les représentations, et leur efficacité varie énormément d'un langage à l'autre.

Efficacité opérationnelle

L'idée fondamentale de la carte de compilation est de comparer les langages selon leur capacité à *supporter* (ou *satisfaire*) des opérations élémentaires, c'est-à-dire leur capacité à rendre ces opérations polynomiales. L'utilisateur, après avoir identifié les opérations dont il a besoin en ligne, peut alors choisir un langage qu'il sait supporter ces opérations ; il est ainsi assuré que la complexité de son application en ligne est polynomiale en la taille des structures manipulées.

L'idée de classifier les langages selon les opérations élémentaires qu'ils supportent a été introduite par Darwiche et Marquis [DM02], dans le cadre propositionnel. Ils ont distingué deux catégories d'opérations : les *requêtes* et les *transformations*. Les requêtes sont des opérations qui renvoient de l'information sur la fonction que représente la forme compilée ; par exemple, dans le cadre propositionnel, « la formule est-elle cohérente ? » ou bien « quels sont les modèles de cette formule ? ». Les transformations sont des opérations qui renvoient un élément du langage considéré ; par exemple, dans le cadre propositionnel, « quelle est la négation de cette formule ? » ou « quelle est la conjonction de ces deux formules ? »

On peut voir que dans les deux cas, les requêtes et les transformations *renvoient quelque chose* ; la distinction entre les deux semble à première vue superficielle. Elle est pourtant très importante : les requêtes sont indépendantes du langage considéré, contrairement aux transformations. En effet (toujours dans le cadre propositionnel), la réponse à la requête « la formule est-elle cohérente ? » est la même que la formule soit représentée par une CNF, une DNF, un OBDD, ou dans n'importe quel autre fragment booléen. Au contraire, le résultat de la transformation « quelle est la négation de cette formule ? » doit être exprimée dans le même langage que la formule elle-même ; si la formule est un BDD, on en veut la négation sous forme de BDD, ce qui est facile, mais si c'est une DNF, on en veut la négation sous forme de DNF, ce qui est difficile.

Nous donnons maintenant une définition formelle de la satisfaction d'une requête et d'une transformation. Chaque opération est associée à une « fonction d'interrogation », qui fournit une réponse (un élement d'un ensemble *Answers*, souvent une valeur booléenne) à une question (qui dépend de paramètres, considérés comme les éléments d'un ensemble *Params*) à propos de certaines représentations d'un langage. Par exemple, pour la requête de « vérification de modèle », *Params* est l'ensemble de toutes les instanciations possibles, et *Answers* $= \mathbb{B}$.

Définition 1.2.16 (Requête et transformation)**.** Soit $\mathfrak{D}$ un domaine d'interprétation, $n \in \mathbb{N}^*$, *Params* et *Answers* des ensembles quelconques, et $f : \mathfrak{D}^n \times \textit{Params} \to \textit{Answers}$. On dit que f est une *fonction d'interrogation*. Soit $\mathtt{L}$ un langage de représentation sur un domaine d'interprétation inclus dans $\mathfrak{D}$.

On dit que $\mathtt{L}$ *satisfait* ou *supporte* la *requête* **Q** associée à f si et seulement s'il existe un polynôme P et un algorithme associant tout n-uplet de $\mathtt{L}$-représentations $\langle \varphi_1, \ldots, \varphi_n \rangle$ et tout élément $\pi \in \textit{Params}$, à un élément $\alpha \in \textit{Answers}$ vérifiant $\alpha = f(\llbracket \varphi_1 \rrbracket_{\mathtt{L}}, \ldots, \llbracket \varphi_n \rrbracket_{\mathtt{L}}, \pi)$, en un temps borné par $P(\|\varphi_1\|, \ldots, \|\varphi_n\|, \|\pi\|, \|\alpha\|)$.

Si *Answers* $= \mathfrak{D}$, on dit que $\mathtt{L}$ *satisfait* ou *supporte* la *transformation* **T** associée à f si et seulement s'il existe un polynôme P et un algorithme associant tout n-uplet de $\mathtt{L}$-représentations $\langle \varphi_1, \ldots, \varphi_n \rangle$ et tout élément $\pi \in \textit{Params}$, à une $\mathtt{L}$-représentation ψ telle que $\llbracket \psi \rrbracket_{\mathtt{L}} = f(\llbracket \varphi_1 \rrbracket_{\mathtt{L}}, \ldots, \llbracket \varphi_n \rrbracket_{\mathtt{L}}, \pi)$, en un temps borné par $P(\|\varphi_1\|, \ldots, \|\varphi_n\|, \|\pi\|)$.

Pour illustrer cela, donnons une définition formelle de la requête de « vérification de modèle » dans ce formalisme. Soit $\mathtt{L}$ un langage de représentation sur le domaine d'interprétation $\mathfrak{D}_{V,\mathbb{B}}$, et soit *Params* l'ensemble de toutes les V-instanciations ; la requête de « *model-checking* » est celle associée à la fonction d'interrogation

$$f : \begin{array}{ccc} \mathfrak{D}_{V,\mathbb{B}} \times \textit{Params} & \to & \mathbb{B} \\ \langle I, \vec{v} \rangle & \mapsto & I(\vec{v}) \end{array} .$$

Faisons ici une remarque fondamentale : pour qu'un langage supporte une requête, la taille de la sortie de l'algorithme ne doit nullement être polynomiale en la taille de l'entrée, puisque le polynôme utilisé pour borner la complexité dépend de l'entrée *et* de la sortie. De cette façon, une requête dont la réponse est généralement de taille exponentielle en la taille de l'entrée (typiquement, demander d'énumérer

les modèles d'une formule propositionnelle) peut tout de même être satisfaite. En revanche, en ce qui concerne les transformations, la taille de la forme compilée de sortie est nécessairement polynomiale en la taille de l'entrée (puisque tout algorithme en temps polynomial est aussi en espace polynomial : $\mathsf{P} \subseteq \mathsf{PSPACE}$ [voir Pap94]).

Bien que n'ayant qu'une définition très générale d'une requête et d'une transformation, on peut d'ores et déjà démontrer certaines propriétés de base.

Proposition 1.2.17. Soit $\mathfrak{D}$ un domaine d'interprétation, et $\mathtt{L}_1$ et $\mathtt{L}_2$ deux langages de représentation de domaine d'interprétation inclus dans $\mathfrak{D}$.

- Si $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, alors les requêtes (définies sur $\mathfrak{D}$) supportées par $\mathtt{L}_1$ sont aussi supportées par $\mathtt{L}_2$.
- Si $\mathtt{L}_1 \sim_p \mathtt{L}_2$, alors $\mathtt{L}_1$ et $\mathtt{L}_2$ supportent exactement les mêmes requêtes et les mêmes transformations (définies sur $\mathfrak{D}$).

Le premier point de cette proposition illustre la distinction entre requêtes et transformations : $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$ est suffisant pour que $\mathtt{L}_2$ supporte les *requêtes* supportées par $\mathtt{L}_1$, car pour obtenir la réponse à une requête **Q** sur une $\mathtt{L}_2$-représentation φ_2, il suffit de calculer une $\mathtt{L}_1$-représentation φ_1 de $[\![\varphi_2]\!]$ (ce qui est faisable en temps polynomial), et de lancer la requête sur φ_1 (ce qui est également faisable en temps polynomial puisque $\mathtt{L}_1$ satisfait **Q**). Néanmoins, il est impossible d'utiliser cette méthode avec les *transformations* : en supposant que $\mathtt{L}_1$ supporte une transformation **T** et que $\mathtt{L}_1 \leqslant_p \mathtt{L}_2$, on peut obtenir en temps polynomial une $\mathtt{L}_1$-représentation de la réponse à la question posée sur une $\mathtt{L}_2$-représentation ; mais cela ne suffit pas, puisque l'on cherche une réponse exprimée dans $\mathtt{L}_2$, et que l'on ne sait pas si on peut obtenir une $\mathtt{L}_2$-représentation en temps polynomial. C'est le cas si on ajoute l'hypothèse que $\mathtt{L}_1 \geqslant_p \mathtt{L}_2$, ce qui explique le second point de la proposition.

Les opérations élémentaires varient selon la catégorie de fonction représentée par le langage-cible considéré. Par exemple, la *cohérence* est une propriété des fonctions booléennes, la *somme arithmétique* une opération sur les fonctions arithmétiques. À première vue, se demander si un ADD [§ 1.1.2] est cohérent ou chercher à faire la somme arithmétique de deux OBDDs n'a pas de sens. Il est pourtant possible de généraliser, en définissant des requêtes et transformations s'appliquant sur n'importe quel langage ; ainsi, le cadre `VNNF` [FM07] contient des requêtes et transformations générales qui capturent bien les notions plus spécifiques de « cohérence » et de « somme arithmétique ». Cependant, la présente thèse ne détaillant que la carte de compilation des langages booléens, nous ne chercherons pas ici à lister toutes les requêtes et transformations à portée générale.

Nous avons achevé la présentation des notions se rapportant au formalisme de la carte de compilation. Cette présentation a été faite dans une optique très générale — les concepts définis peuvent s'appliquer à des langages très variés dans leur interprétation. Dans la section suivante, nous nous limitons aux *langages booléens*, et introduisons un certain nombre de langages existants dans la littérature.

1.3 Langages booléens

On appelle *langages booléens* les langages de représentation $\mathtt{L}$ tels que $E_{\mathtt{L}} = \mathbb{B}$ (sans restriction sur les variables) : le domaine d'interprétation des langages booléens est donc l'espace des fonctions booléennes. Ces langages constituent la partie la plus étudiée de la compilation de connaissances ; cela s'explique par le fait qu'ils sont à la fois très puissants et simples d'utilisation. Notons que les fonctions booléennes peuvent être vues comme des *ensembles d'instanciations*, cela leur permettant notamment d'englober les réseaux de contraintes [§ 1.5.2], vus comme l'ensemble de leurs solutions.

Dans cette section, nous donnons la définition d'un certain nombre de langages booléens de la littérature pour lesquels des résultats de compilation sont connus. Nous laissons de côté les logiques de haut niveau, beaucoup plus compactes, comme celles listées par Gogic et al. [GK$^+$95], et nous limitons à une structure de données spécifique (quoique très générale), celles des graphes acycliques orientés.

1.3.1 Langages de DAGs

De nombreuses représentations des fonctions booléennes sont des graphes acycliques orientés (DAGs, *directed acyclic graphs*) à racine unique, et qui plus est, fonctionnent selon les mêmes mécanismes. Nous définissons ici le langage général des DAGs, que nous appelons `GRDAG` (*general rooted DAG language*), qui comprend tous les langages booléens que nous définirons dans la suite.

Commençons par définir formellement ce qu'est un graphe orienté, ainsi que les concepts associés [voir e.g. Gib85].

Définition 1.3.1 (Graphe orienté)**.** Un *graphe orienté* Γ est un couple $\langle \mathscr{N}_\Gamma, \mathscr{E}_\Gamma \rangle$, où $\mathscr{N}_\Gamma$ est un ensemble fini dont les éléments sont appelés *nœuds* (ou *sommets*), et $\mathscr{E}_\Gamma \subseteq \mathscr{N}_\Gamma^2$ est un ensemble de couples de nœuds, appelés *arcs*.

Soit $E = \langle N_1, N_2 \rangle \in \mathscr{E}_\Gamma$; N_1 est appelé l'*origine de l'arc* E, et N_2 sa *destination* ; N_2 est dit *fils* de N_1, et N_1 *père* de N_2. Les *arcs sortants* d'un nœud N sont ceux ayant N pour origine ; les *arcs entrants* d'un nœud N sont ceux ayant N pour destination.

Soit $N \in \mathscr{N}_\Gamma$; N est une *racine de* Γ si et seulement s'il n'a pas de père. N est une *feuille de* Γ si et seulement s'il n'a pas de fils.

On note $\mathrm{Src}(E)$ (resp. $\mathrm{Dest}(E)$) l'origine (resp. la destination) d'un arc E, $\mathrm{Out}(N)$ (resp. $\mathrm{In}(N)$) l'ensemble des arcs sortants (resp. entrants) d'un nœud N, et $\mathrm{Ch}(N)$ (resp. $\mathrm{Pa}(N)$) l'ensemble des fils (resp. des pères) d'un nœud N. Notons que la définition n'empêche pas un graphe d'être *vide*, c'est-à-dire de ne contenir aucun nœud ; cependant, elle interdit à un graphe d'être infini. En compilation de connaissances, les graphes sont presque toujours *étiquetés*, c'est-à-dire que chaque nœud ou arc est associé à un certain objet mathématique (comme une variable ou une

valeur). Selon notre définition d'un graphe orienté, il ne peut y avoir au plus qu'un seul arc entre deux nœuds donnés ; le fait d'utiliser des étiquettes nous oblige parfois à recourir à des *multigraphes*, qui sont définis de la même manière, avec la différence que $\mathscr{E}_\Gamma$ est un *multi-ensemble fini* de couples de nœuds, ce qui permet de joindre deux nœuds donnés par plusieurs arcs, ayant possiblement des étiquettes différentes.

Pour introduire les DAGs, nous avons besoin des notions classiques de *chemin* et de *cycle*.

Définition 1.3.2 (Chemin et cycle)**.** Soit Γ un graphe (ou multigraphe) orienté, et soit N et N' deux nœuds de Γ. Un *chemin de N à N'* est une suite d'arcs (possiblement vide) $p = \langle E_1, E_2, \ldots, E_k \rangle \in \mathscr{E}_\Gamma^{\circ k}$ telle que

- $\text{Src}(E_1) = N$ et $\text{Dest}(E_k) = N'$;
- $\forall i \in \{1, \ldots, k-1\}, \text{Dest}(E_i) = \text{Src}(E_{i+1})$.

S'il existe un chemin de N à N', N est appelé *ancêtre* de de N', et N' *descendant* de N.

Un *cycle* dans Γ est un chemin *non vide* d'un nœud $N \in \mathscr{N}_\Gamma$ à lui-même.

Nous pouvons maintenant définir les DAGs.

Définition 1.3.3 (DAG)**.** Un *graphe acyclique orienté* (DAG, *directed acyclic graph*) est un graphe orienté n'ayant aucun cycle.

Un DAG est dit *à racine unique* (RDAG, *rooted DAG*) si et seulement s'il est vide ou a exactement une seule racine.

Insistons sur l'importance de la condition imposant aux cycles d'être non vides : sans elle, tout graphe orienté serait trivialement un DAG. Notons également qu'un RDAG non vide a toujours au moins une feuille (qui peut également être racine).

Passons à présent à notre définition d'un langage booléen général se basant sur les graphes. Cette généralisation est inspirée des travaux de Fargier et Marquis [FM07][3].

Définition 1.3.4 (GRDAG)**.** Un GRDAG (*general rooted DAG*) est un RDAG satisfaisant les conditions suivantes :

- chaque feuille est étiquetée par une constante booléenne $\top$ ou $\bot$ (la feuille est alors une *constante*), ou par un couple $\langle x, A \rangle$, où x est une variable et A un ensemble (la feuille est alors un *littéral*) ;
- chaque nœud interne (non-feuille) est soit étiqueté par un opérateur binaire sur $\mathbb{B}$ et a alors deux fils (ordonnés), soit étiqueté par l'opérateur unaire $\neg$ ou un quantificateur $\exists x$ ou $\forall x$ (avec $x \in \mathcal{V}$), et a alors exactement un fils.

[3] Le langage `VNNF` qu'ils définissent est beaucoup plus général, puisqu'il englobe plusieurs formalismes différents, certains d'entre eux hors du cadre de la compilation de connaissances. Cependant, notre langage `GRDAG` n'est pas à proprement parler inclus dans `VNNF`, puisque nous généralisons certains de ses éléments.

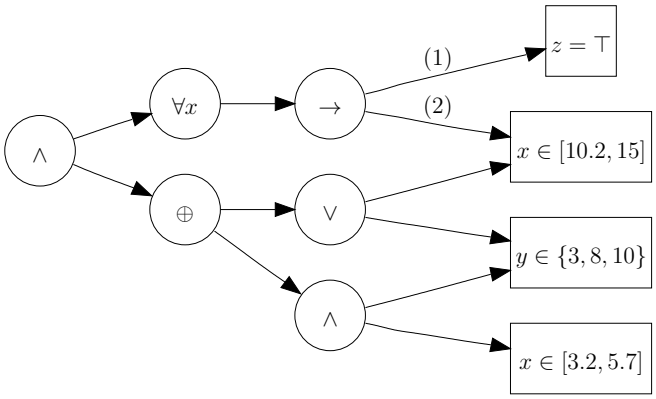


Fig. 1.5 : Un exemple de GRDAG. Il porte sur les variables $x \in \mathcal{V}_{\mathbb{R}}$, $y \in \mathcal{V}_{\mathbb{N}}$ et $z \in \mathcal{B}$.

La figure 1.5 montre un exemple de GRDAG. Notons qu'un nœud peut être étiqueté par n'importe quel opérateur binaire, non seulement « $\vee$ » ou « $\wedge$ », mais également « $\oplus$ » (ou exclusif), « $\rightarrow$ » (implication), la négation de la conjonction ou de la disjonction, etc. En particulier, les opérateurs ne sont pas nécessairement commutatifs ni associatifs. L'ordre des fils doit être déterminé, comme on peut le voir sur l'exemple (nœud d'implication). On note $\mathbb{B}^{\mathbb{B}^2}$ l'ensemble de tous les opérateurs binaires sur $\mathbb{B}$, et Ops l'ensemble $\mathbb{B}^{\mathbb{B}^2} \cup \{\neg, \exists, \forall\}$.

Pour définir la fonction de taille caractéristique des GRDAGs, on commence par introduire la taille d'un nœud N : on décide que $\|N\| = 1$ dans tous les cas, sauf s'il s'agit d'un littéral $\langle x, A\rangle$, auquel cas $\|N\| = \|A\|$ (ce qui dépend donc de la façon dont A est représenté). Alors, pour tout GRDAG φ d'ensemble de nœuds $\mathcal{N}_\varphi$, $\|\varphi\| = \sum_{N \in \mathcal{N}_\varphi} \|N\| + \sum_{x \in \text{Scope}(\varphi)} \|\text{Dom}(x)\|$. Nous devons faire intervenir la taille des domaines des variables ; en effet, nous ne pouvons nous contenter de considérer qu'une structure ne fait que *mentionner* les variables, sans avoir aucune information sur les variables elles-mêmes, car cela compromettrait la polynomialité de certaines opérations importantes, telles que la négation. Cela est cependant sans grande conséquence pratique, car on n'utilise que rarement des domaines « complexes », mais généralement des domaines représentables par un intervalle (spécifié par ses deux bornes).

Définition 1.3.5 (Sémantique d'un GRDAG)**.** L'interprétation et la portée d'un GRDAG φ sont définis récursivement comme suit.

- Si la racine de φ est une feuille étiquetée par $\bot$ (resp. $\top$), alors $\text{Scope}(\varphi) = \varnothing$, et $[\![\varphi]\!]$ est la fonction renvoyant toujours $\bot$ (resp. $\top$).
- Si la racine de φ est une feuille étiquetée par $\langle x, A\rangle$, alors $\text{Scope}(\varphi) = \{x\}$, et $[\![\varphi]\!]$ est la fonction $\text{Dom}(x) \to \mathbb{B}$, telle que $[\![\varphi]\!](\vec{x}) = \top$ si et seulement si $\vec{x} \in A$.

- Si la racine de φ est un nœud interne étiqueté par un opérateur binaire $\otimes$, de fils N_l et N_r, alors (en notant φ_l (resp. φ_r) le GRDAG de racine N_l (resp. N_r)) $\text{Scope}(\varphi) = \text{Scope}(\varphi_l) \cup \text{Scope}(\varphi_r)$, et $[\![\varphi]\!]$ est la fonction $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ qui vérifie, pour toute $\text{Scope}(\varphi)$-instanciation $\vec{v}$, $[\![\varphi]\!](\vec{v}) = [\![\varphi_l]\!](\vec{v}) \otimes [\![\varphi_r]\!](\vec{v})$.

- Si la racine de φ est un nœud interne étiqueté par l'opérateur unaire $\neg$ de fils N_c, alors (en notant φ_c le GRDAG de racine N_c) $\text{Scope}(\varphi) = \text{Scope}(\varphi_c)$, et $[\![\varphi]\!]$ est la fonction $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ qui vérifie, pour toute $\text{Scope}(\varphi)$-instanciation $\vec{v}$, $[\![\varphi]\!](\vec{v}) = \neg[\![\varphi_c]\!](\vec{v})$.

- Si la racine de φ est un nœud interne étiqueté par une quantification $\forall x$ (resp. $\exists x$) de fils N_c, alors (en notant φ_c le GRDAG de racine N_c) $\text{Scope}(\varphi) = \text{Scope}(\varphi_c) \setminus \{x\}$, et $[\![\varphi]\!]$ est la fonction $\text{Dom}(\text{Scope}(\varphi)) \to \mathbb{B}$ qui vérifie, pour toute $\text{Scope}(\varphi)$-instanciation $\vec{v}$, $[\![\varphi]\!](\vec{v}) = \top$ si et seulement si quelle que soit $\vec{x} \in \text{Dom}(x)$, $[\![\varphi_c]\!](\vec{v} \,.\, \vec{x}) = \top$ (resp. il existe $\vec{x} \in \text{Dom}(x)$ telle que $[\![\varphi_c]\!](\vec{v} \,.\, \vec{x}) = \top$).

Il est important de remarquer que les GRDAGs sont très proches des formules logiques classiques ; la différence est qu'au lieu de porter sur des variables booléennes, elles portent sur des « conditions d'appartenance ». Cela rappelle le problème de satisfaisabilité modulo une théorie (SMT), qui consiste à vérifier la cohérence d'une formule logique dans laquelle certaines variables booléennes sont remplacées par des prédicats de théories du premier ordre [voir e.g. Seb07]. De plus, les formules ne sont pas représentées comme des suites de symboles, mais comme des graphes, ce qui permet d'éviter la répétition des sous-formules communes, à la manière de la fusion des sous-graphes isomorphes dans les BDDs [§ 1.1.2.3]. Malgré ces différences, nous appellerons parfois les GRDAGS des « formules ».

En nous servant de ces divers éléments, nous pouvons maintenant définir le langage de représentation `GRDAG`.

Définition 1.3.6. `GRDAG` est le langage de représentation $\langle \mathfrak{D}_{\mathcal{V},\mathbb{B}}, \mathcal{R}, [\![\cdot]\!] \rangle$, où $\mathcal{R}$ est l'ensemble de tous les GRDAGs, et $[\![\cdot]\!]$ est la fonction d'interprétation donnée dans la définition 1.3.5.

Insistons sur la différence de police de caractères entre `GRDAG` (le langage de représentation) et un GRDAG (une `GRDAG`-représentation). Dans la suite, nous utiliserons toujours une police spécifique pour différencier langage et structures, comme c'est la cas dans la carte de compilation originelle [DM02].

On définit trois paramètres sur les GRDAGs qui auront leur importance dans la suite. La *hauteur* d'un GRDAG φ, notée $\text{h}(\varphi)$, est le nombre d'arcs du plus long chemin de la racine à une feuille de φ. On note $\text{Labels}(\varphi)$ l'*ensemble des étiquettes de* φ, c'est-à-dire l'ensemble de tous les ensembles étiquetant un littéral dans φ. On note $\text{Ops}(\varphi)$ l'*ensemble des opérateurs de* φ, c'est-à-dire l'ensemble défini récursivement comme suit :

- $\text{Ops}(\varphi) \subseteq \text{Ops}$;

- Ops(φ) contient $\otimes \in \mathbb{B}^{\mathbb{B}^2} \cup \{\neg\}$ si et seulement si au moins un nœud de φ est étiqueté par $\otimes$;
- Ops(φ) contient $\exists$ (resp. $\forall$) si et seulement si au moins un nœud de φ est étiqueté par $\exists x$ (resp. $\forall x$), où $x \in \mathcal{V}$.

On utilise un certain nombre de simplifications, de manière à clarifier ou alléger les notations. Tout d'abord, on ne fait généralement pas de distinction entre un nœud N et le sous-graphe φ de racine N ; ainsi, Scope(N) correspond simplement à Scope(φ). On ne fait pas non plus de distinction entre un graphe et son interprétation : on appelle par exemple « littéral » un graphe consistant en une simple feuille littérale. On utilisera parfois la formulation « un nœud $\big[x \in A\big] \wedge \big[y \in B\big]$ » pour désigner un nœud $\wedge$ ayant pour fils deux littéraux, étiquetés respectivement par $\langle x, A\rangle$ et $\langle y, B\rangle$. Une autre simplification commune est d'autoriser les nœuds étiquetés par un opérateur commutatif et associatif à avoir plus de deux fils ; cela ne pose aucun problème car la taille caractéristique ne change que logarithmiquement.

Le premier résultat que nous prouvons est que GRDAG est un langage incomplet, à cause de la limitation des littéraux aux contraintes d'appartenance.

Proposition 1.3.7. GRDAG est incomplet.

On peut l'observer par exemple en considérant une fonction $f\colon \mathbb{R}^2 \to \mathbb{B}$ définie par $f\colon \langle x, y\rangle \mapsto \big[x = y\big]$. f peut être représentée par la formule $f = \bigvee_{r\in\mathbb{R}}(\big[x \in \{r\}\big] \wedge \big[y \in \{r\}\big])$, mais celle-ci ne peut être exprimée comme une GRDAG, car nous ne considérons que des graphes finis. Cette incomplétude n'implique pas que les sous-langages de GRDAG soient tous incomplets : comme nous l'avons mentionné plus haut, la complétude dépend fortement du domaine d'interprétation. Ainsi, par exemple, $\text{GRDAG}_{\mathcal{B}}$ est complet.

Le reste de cette section est dédié à la présentation de langages de compilation de la littérature. Cependant, nous les définissons de manière plus générale, en les faisant porter sur des variables potentiellement non-booléennes ou non-énumérées. Nous préciserons dans chaque cas la catégorie de variables sur laquelle le langage portait originellement. Une conséquence de cette généralisation est que les GRDAGs permettent l'utilisation de littéraux représentant $\big[x \in A\big]$ avec A n'étant pas un singleton, et n'étant pas inclus dans Dom(x). Cela nous sera utile dans les chapitres suivants [deuxième partie], mais dans la littérature, tous les langages n'utilisent comme littéraux que des singletons inclus dans le domaine de la variable. Nous avons donc besoin d'introduire une restriction des GRDAGs.

Définition 1.3.8 (Restriction de l'expressivité littérale)**.** Soit $\mathcal{A}$ un ensemble d'ensembles et φ un GRDAG. On dit que *l'expressivité littérale de φ est restreinte à $\mathcal{A}$* si et seulement si Labels(φ) $\subseteq \mathcal{A}$. On note $\text{GRDAG}^{\mathcal{A}}$ le sous-langage de GRDAG d'expressivité littérale restreinte à $\mathcal{A}$.

La principale restriction d'expressivité littérale dont nous aurons besoin pour caractériser les langages existants est la restriction aux littéraux de la forme $\langle x, \top\rangle$ et $\langle x, \bot\rangle$, excluant ceux de la forme $\langle x, \varnothing\rangle$ et $\langle x, \{\bot, \top\}\rangle$. Nous considérerons donc

souvent des fragments de $\texttt{GRDAG}_{\mathcal{B}}^{\mathbb{SB}}$, l'ensemble $\mathbb{SB}$ étant l'ensemble des singletons de $\mathbb{B}$. Parfois, on rencontrera des fragments de $\texttt{GRDAG}_{\mathcal{E}}^{\mathbb{SZ}}$, $\mathcal{E}$ étant l'ensemble des variables énumérées (c'est-à-dire ayant pour domaine un intervalle fini d'entiers) et $\mathbb{SZ}$ étant l'ensemble des singletons de $\mathbb{Z}$.

1.3.2 Fragments basé sur des restrictions d'opérateurs

On obtient divers sous-langages de GRDAG en restreignant les opérateurs étiquetant les nœuds.

Restriction aux opérateurs de base

Commençons avec un fragment ayant une importance particulière en compilation de connaissances, puisqu'il s'agit de la « racine » originelle de la carte de compilation propositionnelle [DM02]. Nommé d'après les formules en forme normale négative de la logique propositionnelle [Bar77], il fut introduit en tant que langage-cible de compilation par Darwiche [Dar01a].

Définition 1.3.9 (NNF)**.** Un GRDAG φ est en *forme normale négative (NNF)* si et seulement si $\text{Ops}(\varphi) \subseteq \{\wedge, \vee\}$.
NNF est la restriction de GRDAG aux formules en NNF.

Le nom « forme normale négative » peut sembler étrange, puisque la définition implique justement de retirer la négation de la liste des opérateurs autorisés. Dans la définition originelle (qui correspond ici au langage $\texttt{NNF}_{\mathcal{B}}^{\mathbb{SB}}$), ce nom fait référence au fait que toutes les négations portent sur les littéraux. Comme nous considérons des NNFs sur des variables non-booléennes, avec des feuilles de la forme $\big[x \in A\big]$, il n'y a plus de rapport avec la négation (le littéral propositionnel $\neg x$ correspond au littéral $\langle x, \{\bot\}\rangle$ dans un GRDAG). Nous conservons néanmoins ce nom pour des raisons de simplicité.

Ajout de la négation

Le fragment PDAG a été défini par Wachter et Haenni [WH06] comme une extension de NNF, lui ajoutant la possibilité d'utiliser des nœuds $\neg$.

Définition 1.3.10 (PDAG)**.** Un GRDAG φ est appelé *DAG propositionnel (PDAG)*, si et seulement si $\text{Ops}(\varphi) \subseteq \{\wedge, \vee, \neg\}$.
PDAG est la restriction de GRDAG aux PDAGs.

L'acception originelle des PDAGs correspond ici aux $\texttt{PDAG}_{\mathcal{B}}^{\mathbb{SB}}$-représentations ne contenant que des littéraux positifs ; il ont été modifiés pour accepter les littéraux négatifs par Fargier et Marquis [FM08a], ladite modification rendant les PDAGs plus proches des NNFs en termes de définition, sans avoir de lourdes conséquences. De manière triviale, on a donc $\texttt{NNF} \subseteq \texttt{PDAG}$.

Ajout de quantificateurs

Le langage suivant comprend la plupart des langages usuels de la compilation de connaissances ; il a été introduit [FM08a] pour tirer parti des principes de fermeture [§ 1.3.7].

Définition 1.3.11 (*Q*PDAG)**.** Un GRDAG φ est un *DAG propositionnel quantifié (QPDAG)* si et seulement si $\mathrm{Ops}(\varphi) \subseteq \{\wedge, \vee, \neg, \exists, \forall\}$.

*Q*PDAG est la restriction de GRDAG aux QPDAGs.

Le langage *Q*PDAG tel que défini par Fargier et Marquis [FM08a] correspond ici à $\textit{Q}\mathtt{PDAG}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$.

1.3.3 Fragments « historiques »

Dans cette section, on retrouve en tant que fragments de NNF certains langages bien connus, qui étaient utilisés avant même la formalisation de l'idée de compilation de connaissances. Ils ont été ajoutés à la carte de compilation par Darwiche et Marquis [DM02] et Fargier et Marquis [FM08b].

Langages plats de NNF

Commençons par généraliser des éléments importants de la logique propositionnelle, les clauses et les termes, pour les intégrer dans le formalisme GRDAG.

Définition 1.3.12 (Clauses et termes)**.** Un nœud d'un GRDAG est une *clause* (resp. un *terme*) si et seulement s'il s'agit d'une constante, d'un littéral, ou d'un nœud interne étiqueté $\vee$ (resp. $\wedge$) dont tous les fils sont des littéraux.

On appelle également clause (resp. terme) un GRDAG dont la racine est une clause (resp. un terme).

On définit term comme la restriction de GRDAG aux termes, et clause comme la restriction de GRDAG aux clauses.

La définition suivante [DM02] introduit les premiers langages de cette carte qui soient efficaces du point de vue des opérations.

Définition 1.3.13 (Platitude et jonction simple)**.** Une NNF φ est *plate* si et seulement si $\mathrm{h}(\varphi) \leqslant 2$.

Une NNF satisfait la propriété de *disjonction simple* (resp. *conjonction simple*) si et seulement si chaque nœud $\vee$ (resp. chaque nœud $\wedge$) est une clause (resp. un terme) dont les fils ne partagent pas de variables.

f-NNF est le fragment de NNF satisfaisant la platitude ; CNF et DNF sont les fragments de f-NNF satisfaisant respectivement la disjonction simple et la conjonction simple.

CNF et DNF sont nommés d'après la forme de leurs représentations ; en effet, les $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ et $\mathtt{DNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$-représentations sont les formules en forme normale conjonctive (CNF) et forme normale disjonctive (DNF) bien connues.

Prime implication

Passons à d'autres sous-ensembles connus des CNFs et DNFs, pour lesquels nous garderons leur définition originelle de langages propositionnels.

Définition 1.3.14 (Impliqués premiers et PI)**.** Une formule en CNF φ est un *impliqué premier* si et seulement si :

- pour toute clause δ_1 vérifiant $\varphi \models \delta_1$, il existe une clause δ_2 dans φ telle que $\delta_2 \models \delta_1$;
- φ ne contient aucun couple de clauses $\langle \delta_1, \delta_2 \rangle$ tel que $\delta_1 \models \delta_2$.

PI est la restriction de $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ aux impliqués premiers.

Définition 1.3.15 (Implicants premiers et IP)**.** Un formule en DNF φ est un *implicant premier* si et seulement si :

- pour tout terme γ_1 vérifiant $\gamma_1 \models \varphi$, il existe un terme γ_2 dans φ tel que $\gamma_1 \models \gamma_2$;
- φ ne contient aucun couple de termes $\langle \gamma_1, \gamma_2 \rangle$ tel que $\gamma_1 \models \gamma_2$.

IP est la restriction de $\mathtt{DNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ aux implicants premiers.

Krom et Horn

Pour faciliter l'introduction des fragments suivants, qui gardent également leur sens booléen d'origine, nous appelons *littéral positif* un littéral de la forme $\big[x = \top\big]$, et *littéral négatif* un littéral de la forme $\big[x = \bot\big]$. Le *complément* d'un littéral $\big[x = \top\big]$ (resp. $\big[x = \bot\big]$) est $\big[x = \bot\big]$ (resp. $\big[x = \top\big]$). On définit à présent les formules classiques de Krom, de Horn, et Horn-renommables, et les incluons dans la carte en suivant Fargier et Marquis [FM08b].

Définition 1.3.16 (Krom, Horn)**.** Une *clause de Krom* est une clause ayant au plus deux littéraux. Une *clause de Horn* est une clause propositionnelle ayant au plus un littéral positif.

Une formule en CNF dont toutes les clauses sont de Krom (resp. de Horn) est appelée une Krom-CNF (resp. une Horn-CNF).

Les langages KROM - C et HORN - C sont les restrictions de $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ aux Krom-CNFs et aux Horn-CNFs, respectivement. Le langage K/H - C est l'union de KROM - C et de HORN - C.

Définition 1.3.17 (Horn-renommabilité)**.** Une CNF propositionnelle φ est *Horn-renommable* si et seulement s'il existe un sous-ensemble $V \subseteq \text{Scope}(\varphi)$ (que l'on appelle un *renommage de Horn pour* φ) tel que si l'on substitue dans φ chaque littéral l vérifiant $\text{Scope}(l) \in V$ par son complément, on obtient une Horn-CNF.

renH - C est le fragment de $\mathtt{CNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ satisfaisant la Horn-renommabilité.

1.3.4 Fragments basés sur des propriétés des nœuds

Cette section regroupe deux propriétés fondamentales des nœuds $\wedge$ et $\vee$, suivant en cela la présentation classique de la carte de compilation propositionnelle.

Darwiche [Dar98] a introduit les NNFs décomposables pour caractériser des diagnostics basés sur la cohérence. Par la suite, elles ont constitué un fragment très influent, occupant une place avantageuse dans la carte de compilation (plus succinct que les langages classiques, et supportant pourtant des opérations fondamentales). Dans un premier temps, la décomposabilité était définie sur les nœuds $\wedge$ uniquement, mais Fargier et Marquis [FM06] ont appliqué cette propriété aux nœuds $\vee$, avant d'étendre la définition à n'importe quel nœud (en renommant la propriété « décomposabilité simple ») [FM07].

Définition 1.3.18 (Décomposabilité)**.** Un nœud N d'un GRDAG est *décomposable* si et seulement si ses fils ne partagent pas de variables, c'est-à-dire que pour tout couple $\langle N_1, N_2\rangle$ de fils de N distincts, $\text{Scope}(N_1) \cap \text{Scope}(N_2) = \varnothing$.

Un GRDAG est *décomposable* si et seulement si tous ses nœuds $\wedge$ sont décomposables.

`DNNF` est le fragment de `NNF` satisfaisant la décomposabilité.

Insistons sur le fait que dans un *graphe* décomposable, seuls les nœuds $\wedge$ doivent être décomposables. Contrairement à la décomposabilité, le déterminisme [Dar01b] est seulement défini sur les nœuds $\vee$.

Définition 1.3.19 (Déterminisme)**.** Un nœud N d'un GRDAG est *déterministe* si et seulement s'il est étiqueté par $\vee$ et ses fils sont deux à deux logiquement contradictoires, c'est-à-dire que pour tout couple $\langle N_1, N_2\rangle$ de fils de N distincts, $[\![N_1]\!] \wedge [\![N_2]\!] \models \bot$.

Un GRDAG est *déterministe* si et seulement si tous ses nœuds $\vee$ sont déterministes.

`d-NNF` est le fragment de `NNF` satisfaisant le déterminisme ; `d-DNNF` est l'intersection de `d-NNF` et de `DNNF`.

1.3.5 La famille des graphes de décision

La section précédente a abordé des propriétés restrictives portant sur un seul nœud à la fois ; nous présentons maintenant des propriétés portant sur des groupes de nœuds. On retrouve ici les fameux BDDs [§ 1.1.2] en tant que NNFs. Nous présentons la famille des graphes de décision, à la manière de Fargier et Marquis [FM06], par des définitions « en cascade » : nous commençons par définir la notion de *nœud d'assignement*, puis celle de *nœud de décision*, qui utilise la précédente, et nous pouvons alors présenter divers langages qui dépendent de certaines propriétés sur les nœuds — par exemple le langages des graphes dont tous les nœuds de décision sont exclusifs, celui des graphes dont tous les nœuds $\wedge$ sont des nœuds d'assignement, etc.

Commençons donc par présenter les nœuds d'assignement [FM06, FM07].

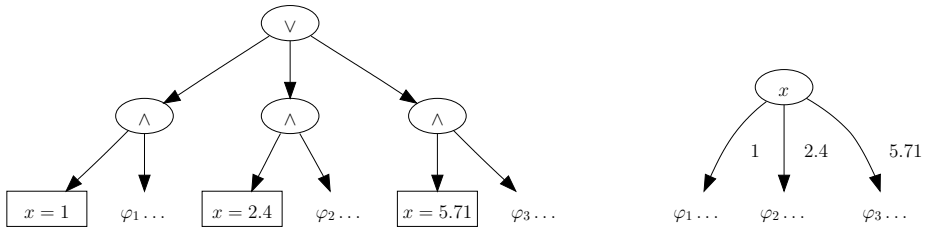


Fig. 1.6 : Un nœud de décision de GRDAG (à gauche) et sa représentation simplifiée (à droite). Sur la version GRDAG, la racine est un nœud de décision, et les trois nœuds $\wedge$ sont des nœuds d'assignement. Dans la représentation simplifiée, le nœud de décision correspond au nœud étiqueté par une variable, et chaque nœud d'assignement correspond à un arc étiqueté.

Définition 1.3.20 (Nœud d'assignment)**.** Un nœud N d'un GRDAG est un *nœud d'assignement* s'il est étiqueté par $\wedge$ et a exactement deux fils, dont exactement un est un littéral.

Un nœud d'assignement N est donc de la forme $\left[x \in A\right] \wedge \alpha$; on dit que N est un nœud d'assignement sur x, et on note $\text{Var}_{\text{ass}}(N)$ la variable x et $\text{Tail}_{\text{ass}}(N)$ le sous-graphe α. Le nom « nœud d'assignement » vient de la définition booléenne d'origine ; ici, de tels nœuds n'assignent pas stricto sensu de valeurs aux variables, mais se contentent de restreindre leurs valeurs possibles. Nous conservons néanmoins le nom original, comme c'est le cas dans le formalisme VNNF [FM07].

On peut maintenant définir les nœuds de décision, toujours d'après Fargier et Marquis [FM07], qui eux-mêmes étendaient les travaux de Darwiche et Marquis [DM02].

Définition 1.3.21 (Nœud de décision)**.** Un nœud N d'un GRDAG est un *nœud de décision* si et seulement s'il est étiqueté par $\vee$ et si tous ses fils sont des nœuds d'assignement sur une même variable x.

Un nœud de décision N est donc de la forme $(\left[x \in A_1\right] \wedge \alpha_1) \vee \dots \vee (\left[x \in A_k\right] \wedge \alpha_k)$; on dit que N est un nœud de décision sur x, et on utilise la notation $\text{Var}_{\text{dec}}(N)$ pour la variable x. La figure 1.6 montre la correspondance entre les nœuds de décision de GRDAG comme nous venons de les définir, et les nœuds de décision tels qu'ils sont classiquement représentés dans le contexte des diagrammes de décision. Les deux représentations sont équivalentes ; nous parlerons respectivement de « version GRDAG » et de « version diagramme de décision ».

On définit les nœuds de décision *exclusifs* en appliquant une restriction aux fils des nœuds de décision.

Définition 1.3.22 (Exclusivité)**.** Deux nœuds d'assignement sur une même variable x d'un GRDAG sont *exclusifs* si et seulement si leurs littéraux respectifs sont exclusifs ; c'est-à-dire qu'en notant $\langle x, A_1 \rangle$ et $\langle x, A_2 \rangle$ les deux littéraux, on a $A_1 \cap A_2 = \varnothing$.

Un nœud de décision est *exclusif* si et seulement si ses fils sont des nœuds d'as-

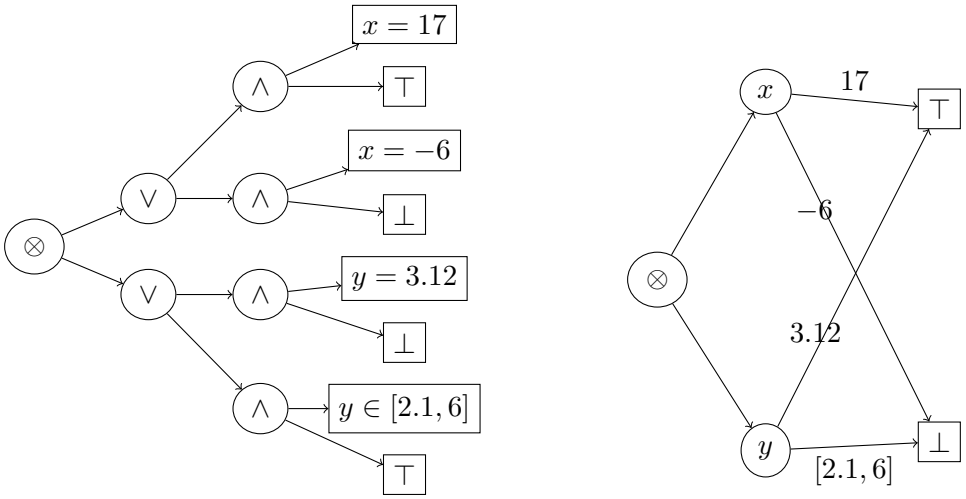


Fig. 1.7 : À gauche, un GRDAG satisfaisant la décision faible. L'étiquette $\otimes$ peut être remplacée soit par $\vee$, soit par $\wedge$; dans les deux cas, la décision faible est satisfaite. À droite, la « version diagramme de décision » de ce GRDAG.

signement exclusifs.

Un GRDAG φ satisfait la propriété de *décision exclusive* si et seulement si tous ses nœuds de décision sont exclusifs.

Si un graphe satisfait la décision exclusive, il est garanti que les branches de chaque nœud de décision soient mutuellement disjointes. Dans la version diagramme de décision (qui utilise des nœuds étiquetés par des variables et des arcs étiquetés), cela implique qu'il ne peut y avoir au plus qu'un seul chemin compatible avec une instanciation donnée : il n'y a pas de choix possible. En particulier, les GRDAGs qui satisfont la décision exclusive sont déterministes — la réciproque n'est cependant pas vraie.

Les propriétés suivantes, qui portent sur la structure des graphes, nous permettent enfin de définir des langages.

Définition 1.3.23 (Propriétés de décision)**.** Soit φ un GRDAG.

- φ satisfait la propriété de *décision faible* si et seulement si :
  - tout littéral de φ a au moins un père, et tous ses pères sont des nœuds d'assignement, et
  - tout nœud d'assignement de φ a au moins un père, et tous ses pères sont des nœuds de décision.
- φ satisfait la propriété de *décision $\vee$-simple* si et seulement s'il satisfait la décision faible, et si tous ses nœuds $\vee$ sont des nœuds de décision.
- φ satisfait la propriété de *décision $\wedge$-simple* si et seulement s'il satisfait la décision faible, et si tous ses nœuds $\wedge$ sont des nœuds d'assignement.

- φ satisfait la propriété de *décision (forte)* si et seulement s'il satisfait à la fois la décision $\vee$-simple et la décision $\wedge$-simple.

Cette définition provient des travaux de Fargier et Marquis [FM06], avec quelques modifications. En particulier, nous avons raffiné la propriété de « décision simple » en trois propriétés.

La décision faible est simplement la condition nécessaire à un GRDAG pour qu'il appartienne à la « famille décisionnelle ». En effet, elle assure qu'il ne contient aucun littéral et aucun nœud d'assignement « libre », qui n'ont pas de représentation dans la « version diagramme de décision ». Par exemple, le GRDAG $\big[x = 1\big] \vee \big[y = 3\big]$ ne satisfait pas la décision faible, car les littéraux ne sont pas fils de nœuds d'assignements. Au contraire, le GRDAG de la figure 1.7 satisfait la décision faible.

En plus de ceci, la décision $\vee$-simple (resp. $\wedge$-simple) garantit qu'aucun nœud $\vee$ (resp. $\wedge$) n'est « libre ». Par exemple, dans le GRDAG de la figure 1.7, si l'opérateur $\otimes$ est un $\wedge$, alors le GRDAG satisfait la décision $\vee$-simple : tous les nœuds $\vee$ sont des nœuds de décision, mais il peut y avoir des nœuds $\wedge$ qui lient ces nœuds de décision. La version diagramme de décision de ces nœuds $\wedge$ « libres » sont simplement des nœuds purement conjonctifs. De la même manière, si l'opérateur $\otimes$ est un $\vee$, alors le GRDAG satisfait la décision $\wedge$-simple : il peut y avoir des nœuds purement disjonctifs dans la version diagramme de décision du GRDAG, mais pas de nœud purement conjonctif.

La condition la plus forte est d'interdire à la fois les nœuds purement disjonctifs et les nœuds purement conjonctifs, c'est-à-dire de satisfaire à la fois la décision $\wedge$-simple et la décision $\vee$-simple : la version diagramme de décision du GRDAG ne contient alors que des nœuds étiquetés par des variables. Nous appelons cette propriété « décision forte » ; dans les travaux d'origine [FM06], elle était simplement appelée « propriété de décision ».

Nous pouvons maintenant définir divers langages de type « graphe de décision », en suivant les travaux susmentionnés.

Définition 1.3.24. Un *graphe de décision* (DG, *decision graph*) est une NNF satisfaisant la décision $\vee$-simple et exclusive. `DG` est la restriction de `NNF` aux graphes de décision.

`DDG` est le fragment de `DG` satisfaisant la décomposabilité.

Un *diagramme de décision basique (BDD)* est une NNF satisfaisant la décision forte et exclusive. `BDD` est la restriction de `NNF` aux BDDs.

Un *free(-ordered) BDD (FBDD)* est un BDD satisfaisant la décomposabilité. `FBDD` est la restriction de `NNF` aux FBDDs.

Notons ici que, comme pour `NNF`, nous gardons les noms des fragments booléens pour des langages qui ne sont pas nécessairement booléens. Ainsi, en utilisant la définition précédente, un BDD peut porter sur des variables non-booléennes : le langage `BDD` classique correspond ici à $\texttt{BDD}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$. À cause de cela, nous avons changé la signification des initiales BDD de « diagramme de décision binaire » en « diagramme de décision basique », car nos BDDs n'ont pas plus de raison d'être qualifiés de « binaires » que n'importe quelle autre représentation de cette carte.

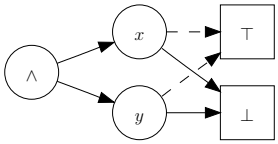


Fig. 1.8 : Un exemple très simple de DG sur des variables booléennes x et y ; le nœud purement conjonctif permet de considérer les deux sous-graphes indépendamment l'un de l'autre.

On peut remarquer que FBDD est le langage des BDDs dans lesquels aucune variable n'est répétée le long d'un chemin (dans la version diagramme de décision des graphes). Cette condition revient à utiliser la décomposabilité [DM02] : elle correspond en effet simplement à exiger que pour chaque nœud d'assignement N, $\text{Var}_{\text{ass}}(N) \notin \text{Scope}(\text{Tail}_{\text{ass}}(N))$, ce qui revient à forcer les portées des fils de N à être disjointes.

Les graphes de décision (les DG-représentations) ajoutent simplement aux diagrammes de décision la possibilité d'utiliser des nœuds purement conjonctifs entre les nœuds de décision, ce qui permet des assignements en parallèle, comme le montre la figure 1.8.

1.3.6 Ordonner les graphes de décision

Ordre général

Cette section s'intéresse à l'ordonnancement des variables dans les graphes de décision, ce qui n'est pas aussi direct que dans les diagrammes de décision binaires de Bryant [Bry86]. Bien que la définition suivante d'un graphe ordonné soit très proche de celle introduite par Bryant, il faut bien noter qu'un ordre n'est ici pas nécessairement total, ce qui permet de tirer parti de la structure « parallèle » des graphes de décision.

Définition 1.3.25 (Graphe ordonné)**.** Soit $<$ un ordre strict sur $V \subseteq \mathcal{V}$; un GRDAG φ est *ordonné par* $<$ si et seulement si $\text{Scope}(\varphi) \subseteq V$ et pour tout couple $\langle N_1, N_2 \rangle$ de nœuds de décision φ tels que N_1 soit un ancêtre de N_2, on a $\text{Var}_{\text{dec}}(N_1) < \text{Var}_{\text{dec}}(N_2)$.

Soit $<$ un ordre strict sur $V \subseteq \mathcal{V}$; O-DDG$_<$ est la restriction de DDG aux graphes ordonnés par $<$. O-DDG est l'union de tous les O-DDG$_<$, pour n'importe quel ordre strict $<$.

Soit $<$ un ordre strict *total* sur $V \subseteq \mathcal{V}$; OBDD$_<$ est la restriction de BDD aux graphes ordonnés par $<$. OBDD est l'union de tous les OBDD$_<$, pour n'importe quel ordre strict total $<$.

$\text{OBDD}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ contient les diagrammes de décision binaires ordonnés tels qu'ils furent introduits par Bryant [Bry86]. Étant donné que $\mathcal{E}$ est l'ensemble des variables énumérées [§ 1.2.1], $\text{OBDD}_{\mathcal{E}}^{\mathbb{S}\mathbb{Z}}$ contient les diagrammes de décision multivalués (MDDs)

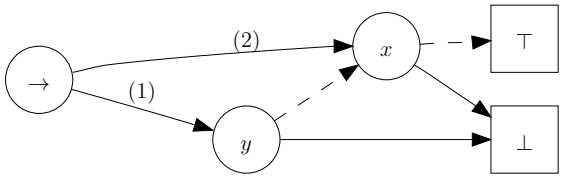


Fig. 1.9 : Un exemple de diagramme d'expression booléenne, avec les nœuds de décision représentées en forme simplifiée. Les variables x et y sont booléennes.

[SK$^+$90] et les automates à états finis [Vem92] — on utilise la notation $\texttt{MDD} = \texttt{OBDD}^{\mathbb{S}\mathbb{Z}}_{\mathcal{E}}$. De manière similaire, on peut retrouver un langage plus exotique, BED, défini par Andersen et Hulgaard [AH97] comme une extension des OBDDs. Sa particularité est qu'il autorise n'importe quel opérateur binaire entre les nœuds de décision ; ce n'est donc pas un fragment de *Q*PDAG. On peut néanmoins utiliser les notions précédemment introduites pour le définir.

Définition 1.3.26 (Diagrammes d'expression booléenne)**.** Un GRDAG φ est un *diagramme d'expression booléenne* (BED, *Boolean expression diagram*) si et seulement si $\mathrm{Ops}(\varphi) \subseteq \mathbb{B}^{\mathbb{B}^2}$, φ satisfait la décision faible et exclusive [déf. 1.3.23], et φ est ordonné par un ordre strict quelconque.

BED est la restriction de GRDAG aux BEDs.

Une exemple simple de BED peut être trouvé en figure 1.9.

Un autre langage est celui des *diagrammes d'intervalles* [ST98], également définis comme une extension des OBDDs. Il s'agit d'un sous-langage d'OBDD, les arcs étant étiquetés par des intervalles d'entiers. Nous ne les définissons pas formellement ici, car cela nécessiterait de nouvelles restrictions spécifiques (notamment sur les arcs sortants des nœuds). Notons cependant que ce langage est contenu dans le langage des *set-labeled diagrams*, que nous introduisons plus tard dans cet ouvrage [chapitre 7].

Ordre fort

O-DDG est très général, mais cette définition de graphe ordonné ne permet pas d'obtenir la canonicité, étant donné que les ordres utilisés ne contraignent pas la façon dont sont utilisés les nœuds purement conjonctifs. Fargier et Marquis [FM06] ont donc défini une restriction de ce langage, autorisant seulement les ordres arborescents.

Définition 1.3.27 (Ordre arborescent)**.** Soit $<$ un ordre strict sur un ensemble X. $<$ est un *ordre arborescent* si et seulement si le graphe $\langle X, <_{\min}\rangle$ est un arbre, avec $<_{\min}$ la plus petite (du point de vue de l'inclusion) relation binaire dont la fermeture transitive soit $<$.

Les ordres arborescents sont donc à mi-chemin entre les ordres stricts et les ordres stricts totaux — puisqu'ils raffinent les ordres stricts et sont raffinés par les ordres stricts totaux. Par exemple, l'ordre strict $<_{\mathsf{a}}$ défini sur $\{x, y, z, t\}$ par $x <_{\mathsf{a}} y <_{\mathsf{a}} z$

et $y <_{\mathrm{a}} t$ est un ordre arborescent, alors que $<_{\mathrm{b}}$ défini par $x <_{\mathrm{b}} y <_{\mathrm{b}} z$ et $t <_{\mathrm{b}} y$ n'en est pas un. Imposons maintenant un ordre arborescent sur les variables.

Définition 1.3.28 (Graphe fortement ordonné)**.** Soit $<$ un ordre arborescent sur $V \subseteq \mathcal{V}$; un GRDAG φ est *fortement ordonné par* $<$ si et seulement si

- φ est ordonné par $<$ (ce qui implique que Scope$(\varphi) \subseteq V$) ;
- pour tout nœud de décision N de φ sur une variable $y \in V$, s'il existe $x \in V$ tel que $x < y$, alors N a un ancêtre qui est un nœud de décision sur x.

Soit $<$ un ordre arborescent sur $V \subseteq \mathcal{V}$; $\mathtt{SO-DDG}_<$ est la restriction de DDG aux graphes fortement ordonnés par $<$. SO-DDG est l'union de tous les $\mathtt{SO-DDG}_<$, pour n'importe quel ordre arborescent $<$.

$\mathtt{SO-DDG}_{\mathcal{C}}^{\mathbb{S}\mathbb{Z}}$ contient les diagrammes de décision multivalués « et/ou » (AOMDDs, *AND/OR multivalued decision diagrams*) [MD06] et les automates arborescents [FV04].

1.3.7 Principes de fermeture

Les principes de fermeture ont été formellement définis par Fargier et Marquis [FM08a]. Ils permettent de définir de nouveaux langages « au-dessus » d'un langage existant, en autorisant l'utilisation d'opérateurs que le langage de base ne contient pas. Nous étendons ici légèrement leurs concepts aux GRDAGs ; commençons par présenter les formules propres.

Définition 1.3.29 (Formule propre)**.** Soit φ un GRDAG, l un littéral dans φ avec Scope$(l) = x$, et Q un nœud étiqueté par une quantification sur x.

On dit que l est *lié par* Q si et seulement s'il existe un chemin du fils de Q à l qui ne contient aucune quantification sur x.

On dit que l est *libre* si et seulement s'il existe un chemin de la racine à l ne contenant aucune quantification sur x.

φ est dit *propre* si et seulement si chacun de ses littéraux est soit libre, soit lié par exactement une quantification sur x. $\mathtt{GRDAG}_p$ est la restriction de GRDAG aux formules propres ; $Q_p\mathtt{PDAG}$ est la restriction de QPDAG aux formules propres.

Le fait de n'utiliser que des formules propres assure qu'aucun littéral ne dépend simultanément de deux quantificateurs, ou n'est à la fois libre et lié. Cela simplifie certaines opérations, qui nécessiteraient la duplication de tels littéraux (par exemple, pendant un conditionnement, les littéraux libres doivent être modifiés, mais pas les littéraux liés, ce qui pose des problèmes si certains littéraux ont les deux propriétés à la fois).

Définition 1.3.30 (Fermeture d'un langage)**.** Soit L un sous-langage de GRDAG, et Δ un sous-ensemble de Ops. La Δ-*fermeture de* L, notée $\mathtt{L}[\Delta]$, est le sous-langage de $\mathtt{GRDAG}_{V_{\mathtt{L}}}$ dont l'ensemble de représentations est défini récursivement comme suit :

- si $\varphi \in \mathtt{L}$, alors $\varphi \in \mathtt{L}[\Delta]$;

- si $\otimes \in \Delta \cap \mathbb{B}^{\mathbb{B}^2}$, φ_l et φ_r sont des L-représentations, alors $\varphi_l \otimes \varphi_r \in \mathtt{L}[\Delta]$;
- si $\neg \in \Delta$ et $\varphi \in \mathtt{L}$, alors $\neg\varphi \in \mathtt{L}[\Delta]$;
- si $\mathrm{q} \in \Delta \cap \{\exists, \forall\}$, $\varphi \in \mathtt{L}$, et $x \in V_{\mathtt{L}}$, alors $\mathrm{q}x.\varphi \in \mathtt{L}[\Delta]$.

Les fermetures sont particulièrement utiles sur les langages incomplets, puisqu'elles peuvent permettre d'en tirer un langage complet tout en maintenant la complexité des principales opérations, comme nous le verrons dans la section 1.4.3. On y considérera les fermetures suivantes, introduites par Fargier et Marquis [FM08b] : KROM - C$[\vee]$, HORN - C$[\vee]$, K/H - C$[\vee]$ et renH - C$[\vee]$.

1.4 Carte de compilation des langages booléens

Cette section résume les résultats connus concernant la carte de compilation des divers langages booléens (c'est-à-dire dont l'ensemble de valuation est $E = \mathbb{B}$) introduits dans la précédente section. Commençons par présenter les requêtes et transformations utilisées pour classer les langages booléens. Elles viennent de la littérature sur la carte de compilation, avec quelques modifications permettant de les adapter à notre contexte impliquant des variables non booléennes.

1.4.1 Requêtes et transformations booléennes

Notions sémantiques

On étend ici certaines notions de la logique — modèle, cohérence, validité, etc. — aux fonctions booléennes en général.

Définition 1.4.1 (Modèle et contre-modèle)**.** Soit f une fonction booléenne sur $V \subseteq \mathcal{V}$; une V-instanciation $\vec{v}$ est un *modèle* (resp. *contre-modèle*) de f, ce que l'on note $\vec{v} \models f$ (resp. $\vec{v} \not\models f$), si et seulement si $f(\vec{v}) = \top$ (resp. $f(\vec{v}) = \bot$).

L'*ensemble des modèles* de f est noté $\mathrm{Mod}(f) \subseteq \mathrm{Dom}(V)$; l'*ensemble de ses contre-modèles* est noté $\mathrm{Mod}^{\mathrm{c}}(f)$. Il est assez clair que $\{\mathrm{Mod}(f), \mathrm{Mod}^{\mathrm{c}}(f)\}$ est une partition de $\mathrm{Dom}(V)$.

Nous étendons cette définition d'un modèle à toute instanciation $\vec{x}$ des variables d'un quelconque ensemble $X \subseteq \mathcal{V}$: en notant $Y = V \setminus X$, $\vec{x}$ est un modèle de f si et seulement si pour toute Y-instanciation $\vec{y}$, $f(\vec{x}_{|V} \,.\, \vec{y}) = \top$. On écrit également $\vec{x} \models f$, et on étend de la même façon la notion de contre-modèle. Il est à noter que les ensembles de modèles et de contre-modèles ne contiennent toujours que des V-instanciations.

Définition 1.4.2 (Cohérence et validité)**.** Soit f une fonction booléenne sur $V \subseteq \mathcal{V}$. f est *cohérente* si et seulement si $\mathrm{Mod}(f) \neq \varnothing$. f est *valide* si et seulement si $\mathrm{Mod}^{\mathrm{c}}(f) = \varnothing$.

Définition 1.4.3 (Contexte)**.** Soit f une fonction booléenne sur $V \subseteq \mathcal{V}$, $x \in \mathcal{V}$ une variable, et $\omega \in \text{Dom}(x)$. ω est une *valeur cohérente* pour x dans f si et seulement s'il existe une $V \cup \{x\}$-instanciation $\vec{v}$ telle que $\vec{v}_{|x} = \omega$ et $\vec{v} \models f$.

L'ensemble de toutes les valeurs cohérentes pour x dans f est appelé *contexte* de x dans f, et noté $\text{Ctxt}_f(x)$.

Les valeurs cohérentes sont parfois appelées *GAC* (*generalized arc consistent*, arc-cohérentes de manière généralisée) ou *globalement cohérentes* en programmation par contraintes [voir e.g. LS06].

Définition 1.4.4 (Implication et équivalence)**.** Soit f, g des fonctions booléennes. f *implique* g, ce que l'on note $f \models g$, si et seulement si tout modèle de f est un modèle de g. f est *équivalente* à g, ce que l'on note $f \equiv g$, si et seulement si $f \models g$ et $g \models f$.

Toutes les notions précédemment introduites peuvent être définies de manière très simple sur des *représentations*, c'est-à-dire des éléments d'un langage quelconque, par le biais de sa fonction d'interprétation.

Définition 1.4.5. Soit $\mathtt{L}$ un langage de représentation booléen, et soit φ et ψ des $\mathtt{L}$-représentations.

- Une $\text{Scope}(\varphi)$-instanciation $\vec{v}$ est un *modèle* (resp. un *contre-modèle*) de φ, ce que l'on note $\vec{v} \models \varphi$ (resp. $\vec{v} \not\models \varphi$), si et seulement si elle est modèle (resp. contre-modèle) de $[\![\varphi]\!]$.
- On écrit $\text{Mod}(\varphi) = \text{Mod}([\![\varphi]\!])$ et $\text{Mod}^c(\varphi) = \text{Mod}^c([\![\varphi]\!])$.
- φ est *cohérente* si et seulement si $[\![\varphi]\!]$ est cohérente.
- φ est *valide* si et seulement si $[\![\varphi]\!]$ est valide.
- $\varphi \models \psi$ si et seulement si $[\![\varphi]\!] \models [\![\psi]\!]$.
- $\varphi \equiv \psi$ si et seulement si $[\![\varphi]\!] \equiv [\![\psi]\!]$.
- Une *valeur cohérente* pour x dans φ est définie comme une valeur cohérente pour x dans $[\![\varphi]\!]$. On note $\text{Ctxt}_\varphi(x) = \text{Ctxt}_{[\![\varphi]\!]}(x)$.

Les définitions suivantes ont trait aux opérations sur les fonctions booléennes.

Définition 1.4.6 (Opérations binaires)**.** Soit $\otimes \in \mathbb{B}^{\mathbb{B}^2}$ un opérateur binaire sur $\mathbb{B}$, et f, g des fonctions booléennes sur $V_f \subseteq \mathcal{V}$ et $V_g \subseteq \mathcal{V}$, respectivement. La fonction booléenne $f \otimes g$ est définie sur les variables de $V_f \cup V_g$ par $\forall \vec{x} \in \text{Dom}(V_f \cup V_g), (f \otimes g)(\vec{x}) = f(\vec{x}_{|V_f}) \otimes g(\vec{x}_{|V_g})$.

Définition 1.4.7 (Négation)**.** Soit f une fonction booléenne sur $V \subseteq \mathcal{V}$; la fonction booléenne $\neg f$ est définie sur les variables de V par $\forall \vec{x} \in \text{Dom}(V), (\neg f)(\vec{x}) = \neg f(\vec{x})$.

Définition 1.4.8 (Quantification)**.** Soit f une fonction booléenne sur $V \subseteq \mathcal{V}$, et $X \subseteq \mathcal{V}$ un ensemble de variables. La *quantification existentielle de f par X* (également appelée *oubli de X dans f* et *projection existentielle de f sur $V \setminus X$*), notée $\exists X.f$, est la fonction booléenne définie sur $Y = V \setminus X$ par

$$\forall \vec{y} \in \text{Dom}(Y), \quad (\exists X.f)(\vec{y}) = \top \iff \exists \vec{x} \in \text{Dom}(X), f(\vec{y} \,.\, \vec{x}_{|V}) = \top.$$

La *quantification universelle de f par X* (également appelée *enforcement de X dans f* et *projection universelle de f sur $V \setminus X$*), notée $\forall X.f$, est la fonction booléenne définie sur $Y = V \setminus X$ par

$$\forall \vec{y} \in \text{Dom}(Y), \quad (\forall X.f)(\vec{y}) = \top \iff \forall \vec{x} \in \text{Dom}(X), f(\vec{y} \,.\, \vec{x}_{|V}) = \top.$$

Définition 1.4.9 (Restriction)**.** Soient f, g deux fonctions booléennes sur $V_f \subseteq \mathcal{V}$ et $V_g \subseteq \mathcal{V}$, respectivement. La *restriction* de f à g, notée $f_{|g}$, est la fonction booléenne définie sur $Y = V_f \setminus V_g$ par $f_{|g} = \exists V_g.(f \wedge g)$.

Soit $V \subseteq \mathcal{V}$, et soit $\vec{v}$ une V-instanciation. La *restriction* de f à $\vec{v}$, notée $f_{|\vec{v}}$, est la fonction booléenne définie sur $Y = V_f \setminus V$ par $f_{|\vec{v}}(\vec{y}) = f(\vec{y} \,.\, \vec{v})$.

Il est à noter que la restriction à une instanciation peut se voir comme un cas particulier de la restriction à une fonction. L'opération de restriction est utile pour composer deux fonctions ou fixer la valeur de certaines variables. On utilise également la notation $f_{|x=\omega}$ pour la restriction de f à l'instanciation de x à ω.

Requêtes sur les langages booléens

Nous définissons à présent les requêtes standard utilisées pour comparer l'efficacité des langages booléens. Introduites par Darwiche et Marquis [DM02], elles ont été généralisées aux langages de représentation par Fargier et Marquis [FM09], de qui nous adaptons les définitions suivantes.

Nous avons identifié trois types de requêtes ; commençons par la première catégorie, celle des requêtes communes à tous les langages booléens.

Définition 1.4.10 (Requêtes générales)**.** Soit L un langage de représentation booléen.

- L satisfait **CO** (cohérence) (resp. **VA**, validité) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ à 1 si φ est cohérente (resp. valide), et à 0 sinon.
- L satisfait **MC** (*model checking*, vérification de modèle) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et toute instanciation $\vec{v}$ des variables de Scope(φ) à 1 si $\vec{v} \models \varphi$, et à 0 sinon.
- L satisfait **EQ** (équivalence) (resp. **SE**, *sentential entailment*, implication de formules) si et seulement s'il existe un algorithme polynomial associant tout couple de L-représentations $\langle \varphi, \psi \rangle$ à 1 si $\varphi \equiv \psi$ (resp. $\varphi \models \psi$), et à 0 sinon.

Ces requêtes sont des extensions directes des définitions classiques [DM02] aux langages portant sur des variables non-booléennes. Grâce à leur structure simple,

ces requêtes s'appliquent à tout langage booléen. Cela est dû au fait que, contrairement aux requêtes qui suivent, leurs entrées et sorties sont très simples et ne posent aucune ambiguïté quant à la façon dont elles sont représentées.

Les requêtes de la seconde catégorie sont problématiques de ce point de vue : leur sortie est également élémentaire (il s'agit d'un valeur booléenne), mais leurs paramètres sont des objets « complexes », à savoir des clauses et des termes — qui ne sont définis que sur GRDAG. C'est pour cette raison que nous avons choisi de ne définir ces requêtes que sur les sous-langages de GRDAG.

Définition 1.4.11 (Requêtes à paramètres complexes)**.** Soit L un sous-langage de GRDAG.

- L satisfait **CE** (*clausal entailment*, implication clausale) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et toute clause γ dans L à 1 si $\varphi \models \gamma$, et à 0 sinon.
- L satisfait **IM** (vérification d'implicant) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et tout terme γ dans L à 1 si $\gamma \models \varphi$, et à 0 sinon.

Dans cette définition, φ et γ sont des GRDAGs, et en particulier sont toutes deux des L-représentations (pour le même L). Si ce n'était pas le cas, certains problèmes liés à l'expressivité obscurciraient les raisons pour lesquelles un langage supporte ou non ces requêtes, alors même que le but de la carte de compilation est d'exprimer la puissance *intrinsèque* de chaque langage. Par exemple, pour que $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$ satisfasse **CE**, l'algorithme doit être capable de déterminer si une $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$-représentation implique $\big[x = \top\big] \vee \big[y = \bot\big]$, mais *pas* si elle implique des "clauses" non-GRDAG, telles que $\big[x = y\big] \vee \big[y = z\big]$, qui ont une expressivité complètement différente.

Les requêtes de la troisième catégorie n'ont pas de paramètres complexes, mais leurs sorties (liées à la notion d'ensemble de modèles) ne sont pas des valeurs booléennes. Nous ne pouvons pas simplement étendre les requêtes de la littérature ; en effet, le nombre de modèles étant possiblement infini, il est inenvisageable de les compter ou de les énumérer. En ce qui concerne la requête d'« énumération de modèles », nous avons dû décider d'une représentation arbitraire de l'ensemble de modèles. Nous avons choisi d'exiger qu'il soit retourné sous la forme de DNFs, qui sont une façon naturelle de représenter des ensembles continus d'instanciations — dans ce contexte, elles sont appelées *unions de boîtes*. Ce fut plus délicat pour la requête de « comptage de modèles » ; nous avons envisagé la possibilité de la baser sur la taille caractéristique de la DNF-représentation *minimale*, mais prouver la minimalité n'est pas trivial. Au final, nous avons choisi de garder le sens originel de « comptage », et de simplement retourner le nombre de modèles s'il est fini, et ∞ sinon.

Définition 1.4.12 (Requêtes à sortie complexe)**.** Soit L un sous-langage de GRDAG, sur un ensemble de variables $V_{\mathtt{L}}$.

- L satisfait **ME** (*model enumeration*, énumération de modèles) si et seulement s'il existe un polynôme $P(\cdot,\cdot)$ et un algorithme associant toute L-représentation φ à une $\mathtt{DNF}_{V_\mathtt{L}}$-représentation ψ de $[\![\varphi]\!]$ en temps $P(\|\varphi\|,\|\psi\|)$.
- L satisfait **ME^c** (*counter-model enumeration*, énumération de contre-modèles) si et seulement s'il existe un polynôme $P(\cdot,\cdot)$ et un algorithme associant toute L-représentation φ à une $\mathtt{DNF}_{V_\mathtt{L}}$-représentation ψ de $\neg[\![\varphi]\!]$ en temps $P(\|\varphi\|,\|\psi\|)$.
- L satisfait **CT** (comptage de modèles) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ à $|\mathrm{Mod}(\varphi)|$ s'il est fini, et à ∞ sinon.

Une fois de plus, ces requêtes ne sont définies que sur les sous-langages de GRDAG, à cause de problèmes d'expressivité. En effet, pour que leur satisfaction ne soit pas « faussée », il est nécessaire qu'une $\mathtt{DNF}_{V_\mathtt{L}}$-représentation de φ existe, or ce n'est pas garanti si φ n'est pas un GRDAG. Considérons par exemple la fonction booléenne f définie sur les variables réelles x et y par $f(x,y) = \top \iff x = y$; il n'existe aucune GRDAG-représentation de f, et par conséquent aucune DNF-représentation. Pour pouvoir utiliser DNF comme langage-cible de représentation de l'ensemble de modèles, nous devons obliger le langage d'entrée à avoir une expressivité compatible avec DNF — ce qui est le cas de GRDAG.

Avant de passer aux transformations, notons que nous n'avons défini que les requêtes « classiques » ; nous introduisons de nouvelles requêtes ultérieurement [§ 3.3.3].

Transformations sur les langages booléens

Nous présentons ici les transformations standard sur les langages booléens, également introduites par Darwiche et Marquis [DM02] et généralisées aux langages de représentation par Fargier et Marquis [FM09].

Définition 1.4.13 (Transformations)**.** Soit L un langage de représentation booléen sur un ensemble de variables $V_\mathtt{L}$.

- L satisfait **CD** (conditionnement) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et toute instanciation $\vec{v}$ des variables de $V_\mathtt{L}$ à une L-représentation de la restriction $[\![\varphi]\!]_{|\vec{v}}$ de $[\![\varphi]\!]$ à $\vec{v}$.
- L satisfait **FO** (*forgetting*, oubli) (resp. **SFO**, *single forgetting*, oubli simple) si et seulement s'il existe un algorithme polynomial qui associe toute L-représentation φ et tout ensemble (resp. singleton) $X \subseteq V_\mathtt{L}$ à une L-représentation de $\exists X.[\![\varphi]\!]$.
- L satisfait **EN** (*ensuring*, enforcement) (resp. **SEN**, *single ensuring*, enforcement simple) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et tout ensemble (resp. singleton) $X \subseteq V_\mathtt{L}$ à une L-représentation de $\forall X.[\![\varphi]\!]$.

- L satisfait $\wedge$**C** (*closure under* $\wedge$, fermeture pour la conjonction) (resp. $\vee$**C**, *closure under* $\vee$, fermeture pour la disjonction) si et seulement s'il existe un algorithme polynomial associant tout ensemble fini Φ de L-représentations à une L-représentation de $\bigwedge_{\varphi\in\Phi}[\![\varphi]\!]$ (resp. $\bigvee_{\varphi\in\Phi}[\![\varphi]\!]$).
- L satisfait $\wedge$**BC** (*closure under binary* $\wedge$, fermeture pour la conjonction binaire) (resp. $\vee$**BC**, *closure under binary* $\vee$, fermeture pour la disjonction binaire) si et seulement s'il existe un algorithme polynomial associant tout couple $\langle\varphi,\psi\rangle$ de L-représentations à une L-représentation de $[\![\varphi]\!]\wedge[\![\psi]\!]$ (resp. $[\![\varphi]\!]\vee[\![\psi]\!]$).
- L satisfait $\neg$**C** (*closure under* $\neg$, fermeture pour la négation) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ à une L-représentation de $\neg[\![\varphi]\!]$.

La seule transformation délicate à généraliser aux variables non-booléennes était la première. Dans sa définition originelle [DM02], le conditionnement est une "restriction à un terme" ; mais comme dans ce contexte tous les littéraux ont la forme « $x = \top$ » ou « $x = \bot$ », il revient à une restriction à une instanciation. Pour le généraliser aux langages booléens à variables non-booléennes, nous avions le choix entre étendre la définition « restriction à un terme » ou garder l'usage « restriction à une instanciation ». Nous avons choisi la deuxième alternative (qui a l'avantage de ne nécessiter aucun paramètre dépendant du langage de φ — voir la discussion à propos de la définition 1.4.11), et avons étendu la définition originelle comme une nouvelle transformation, appelée explicitement *restriction à un terme* [voir § 3.3.3].

1.4.2 Résultats de compacité

La figure 1.10 présente les résultats de compacité de certains langages que nous avons décrits, sous la forme d'un *graphe de compacité*.

Théorème 1.4.14. Les relations de compacité présentés figure 1.10 sont démontrés.

Démonstration. Provient de diverses sources [DM02, FM08b, FM06]. □

Un graphe de compacité permet de déterminer rapidement la compacité relative d'un groupe de langages. Ayant identifié les langages supportant les opérations désirées, l'utilisateur n'a plus qu'à choisir les plus proches de la racine. Bien sûr, si des opérations « difficiles » sont nécessaires (comme la vérification d'équivalence), on ne peut éviter les langages situés en bas du graphe.

1.4.3 Satisfaction de requêtes et de transformations

Les tableaux 1.1 et 1.2 indiquent quelles requêtes et transformations sont satisfaites par les langages que nous avons présentés.

Théorème 1.4.15. Les résultats des tableaux 1.1 et 1.2 sont démontrés.

Démonstration. Les résultats proviennent de diverses sources [DM02, FM08b, FM06]. □

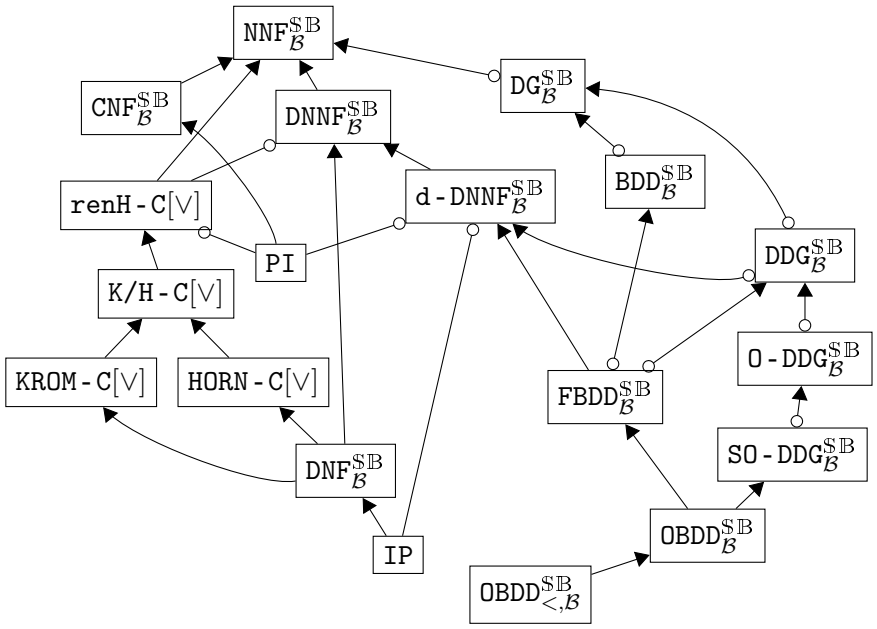


Fig. 1.10 : Compacité de quelques fragments de $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Sur un arc reliant $\mathtt{L}_1$ à $\mathtt{L}_2$, une flèche pointant vers $\mathtt{L}_1$ signifie que $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$. S'il n'y a aucun symbole du côté de $\mathtt{L}_1$ (ni flèche, ni cercle), cela signifie que $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. Un cercle du côté de $\mathtt{L}_1$ signifie qu'on ignore si $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ ou si $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. Les relations que l'on peut déduire par transitivité ne sont pas représentées, ce qui implique que deux fragments n'étant pas ancêtres l'un de l'autre sont incomparables sur le plan de la compacité.

L	CO	VA	CE	IM	EQ	SE	CT	ME
$\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	○	○	○	○	√
$\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	√	○	√	○	○	○	○
PI	√	√	√	√	√	√	○	√
IP	√	√	√	√	√	√	○	√
KROM-C	√	√	√	√	√	√	○	√
KROM-C[∨]	√	○	√	○	○	○	○	√
HORN-C	√	√	√	√	√	√	○	√
HORN-C[∨]	√	○	√	○	○	○	○	√
K/H-C	√	√	√	√	√	√	○	√
K/H-C[∨]	√	○	√	○	○	○	○	√
renH-C	√	√	√	√	√	√	○	√
renH-C[∨]	√	○	√	○	○	○	○	√
$\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	○	○	○	○	√
$\mathtt{d\text{-}NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{d\text{-}DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{DG}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	○	○	√	√
$\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$	○	○	○	○	○	○	○	○
$\mathtt{FBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{O\text{-}DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{O\text{-}DDG}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	?	√	√
$\mathtt{SO\text{-}DDG}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	?	○	√	√
$\mathtt{SO\text{-}DDG}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	?	√	√
$\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	○	√	√
$\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	√	√	√	√	√

Tab. 1.1 : Requêtes satisfaites par certains fragments de $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Les symboles sont les suivants : √ signifie « satisfait », ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ », et ? indique un résultat inconnu.

Cette présentation classique de la carte de compilation se veut facile à lire ; vérifier si un langage satisfait telle requête ou telle transformation est immédiat, et réciproquement, on peut trouver directement l'ensemble des langages satisfaisant telle requête ou telle transformation. Pour remplir cet objectif de lisibilité, les tableaux ne donnent pas d'information précise sur la classe de complexité de chaque problème, mais se contentent d'indiquer si la requête ou la transformation est satisfaite ou non.

1.5 Compilation

Après cette présentation théorique de la compilation de connaissances, incluant de nombreux exemples de langages de représentation et de leurs propriétés, se

L	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
$\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
$\mathtt{DNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	•	√	√	√	•
$\mathtt{CNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	•	√	•
PI	√	√	√	•	•	•	√	•
IP	√	•	•	•	√	•	•	•
KROM-C	√	√	√	√	√	!	!	!
KROM-C[∨]	√	√	√	○	√	√	√	•
HORN-C	√	•	√	√	√	!	!	!
HORN-C[∨]	√	?	√	○	√	√	√	•
K/H-C	√	•	√	○	○	!	!	!
K/H-C[∨]	√	?	√	○	○	√	√	•
renH-C	√	•	√	!	!	!	!	!
renH-C[∨]	√	?	√	○	○	√	√	○
$\mathtt{DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	√	√	○	○	√	√	○
$\mathtt{d\text{-}NNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
$\mathtt{d\text{-}DNNF}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	○	○	○	○	○	?
$\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	○	√	√	√	√	√	√
$\mathtt{FBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	•	○	•	○	•	○	√
$\mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$	√	•	√	•	○	•	○	√
$\mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}$	√	•	√	•	√	•	√	√

Tab. 1.2 : Transformations satisfaites par certains fragments de $\mathtt{NNF}_{\mathcal{B}}^{\$\mathbb{B}}$. Les symboles sont les suivants : √ signifie « satisfait », • signifie « ne satisfait pas », ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ », ! indique que la transformation n'est pas toujours faisable en restant dans fragment, et ? indique un résultat inconnu.

pose la question de l'*utilisation pratique*. Sur le plan théorique, la manipulation des formes compilées est simple : elle consiste à combiner des requêtes et transformations élémentaires. Mais comment obtenir des formes compilées ? En quoi consiste l'étape de compilation proprement dite ? Cette section vise à répondre à ces questions.

1.5.1 Langage d'entrée

Prenons l'exemple d'une personne — que nous appellerons Alice — qui voudrait exécuter un programme sur un système autonome, programme nécessitant la résolution en ligne d'un problème de décision. Elle décide d'utiliser la compilation de connaissances, après avoir remarqué que son problème pouvait se voir comme l'application d'opérations élémentaires sur une large base de connaissances fixée. En s'appuyant sur la carte de compilation, elle identifie le langage-cible booléen L qui correspond le mieux à son application. Elle doit à présent *compiler* son problème en une L-représentation.

Étant destinés à être efficacement manipulable par des machines, les langages-cibles de compilation sont généralement difficiles à manipuler « à la main ». Dar-

wiche et Marquis [DM02] ont distingué deux catégories de langages, ceux développés pour permettre à des humains de coder les connaissances directement, et ceux développés dans l'optique d'être efficaces[4]. Il est fort probable que le problème d'Alice soit représenté dans un langage « naturel » ou « humainement manipulable ». L'étape de *compilation*, en pratique, est donc la traduction depuis ce langage naturel vers le langage-cible qu'elle a identifié.

Cette étape de compilation dépend bien sûr des deux langages en question ; mais cela ne nous empêche pas d'identifier des méthodes générales de compilation. Pour ce faire, nous considérons un langage d'entrée unique, assez général pour englober la plupart des utilisations réelles, celui des *réseaux de contraintes*.

Définition 1.5.1. Un *réseau de contraintes* (CN, *constraint network*) est un couple $\Pi = \langle V, \mathscr{C} \rangle$, où $V \subseteq \mathcal{V}$ est un ensemble de variables (noté $\text{Scope}(\Pi)$) et $\mathscr{C}$ est un ensembles de *contraintes*. Chaque contrainte $C \in \mathscr{C}$ a une portée, notée $\text{Scope}(C)$, et consiste en un ensemble de $\text{Scope}(C)$-instanciations : ce sont les instanciations *autorisées* par C.

Une *solution* de Π est une V-instanciation $\vec{v}$ qui est compatible avec toutes les contraintes : $\forall C \in \mathscr{C}, \vec{v}_{\mid \text{Scope}(C)} \in C$.

L'ensemble de toutes les solutions de Π est appelé son *ensemble solution*, et est noté $\text{Sol}(\Pi)$.

Cette définition est très générique ; en particulier, on peut exprimer les contraintes de diverses manières. La plus directe est d'énumérer explicitement les tuples autorisés ; cela n'est évidemment possible que si les variables concernées ont toutes un domaine fini. La façon la plus naturelle est d'utiliser directement des fonctions et relations de haut niveau sur les variables, ce qui donne des contraintes comme $x = y$, $z \rightarrow x \wedge y$ (sur des variables booléennes)[5], $3x + 2y > z$ (sur des variables numériques), ou même des contraintes globales comme $\text{alldiff}(x_1, \ldots, x_k)$, qui signifie « les variables $x_1, \ldots, x_k$ doivent avoir des valeurs distinctes deux à deux ». Le problème consistant à répondre à la question « Π a-t-il au moins une solution ? » est un problème fondamental en intelligence artificielle, qui porte le nom de *problème de satisfaction de contraintes* (CSP, *constraint satisfaction problem*)[6].

Les réseaux de contraintes permettent d'exprimer de manière très naturelle une base de connaissances, et donc une fonction booléenne (en identifiant les solutions d'un CN aux modèles d'une fonction booléenne). Nous étudions maintenant des méthodes pour traduire un réseau de contraintes vers un langage booléen.

[4] Ils parlent des premiers comme de *langages de représentation* et des seconds comme de *langages-cibles de compilation* — nous ne reprenons pas cette terminologie dans cette thèse, notre définition de « langage de représentation » couvrant les deux acceptations.

[5] Les réseaux de contraintes peuvent donc être utilisées pour représenter des bases de connaissances constituées de formules logiques.

[6] La terminologie « réseau de contraintes » est parfois utilisée pour des contraintes uniquement binaires ; notre définition n'implique nullement une telle restriction — on utilise « réseau de contraintes » pour désigner la structure associée à un CSP, comme le font par exemple Lecoutre et Szymanek [LS06].

1.5.2 Méthodes exactes de compilation

Combinaison de structures élémentaires

La manière classique de compiler un réseau de contraintes vers un langage booléen est souvent appelée *méthode bottom-up*. Elle consiste à combiner des représentations « élémentaires », en appliquant les transformations correspondant à des opérateurs binaires, comme $\wedge_{\mathbf{BC}}$ (la conjonction) ou $\neg_{\mathbf{C}}$ (la négation). Par exemple, la contrainte $(x \wedge y) \rightarrow z$ peut se compiler en appliquant l'opérateur de *conjonction* aux L-représentations de $\left[x = \top\right]$ et $\left[y = \top\right]$, puis en appliquant l'opérateur d'*implication* au résultat et à la L-représentation de $\left[z = \top\right]$. Une fois que chaque contrainte a été compilée, on peut obtenir une représentation du CN tout entier en faisant la conjonction des formes compilées de toutes les contraintes.

Cette méthode ne marche pas avec tous les langages-cibles ; l'application d'opérateur doit être assez facile pour que la forme compilée ne grossisse pas exponentiellement pendant la construction. La méthode *bottom-up* est donc typiquement utilisée pour compiler des diagrammes de décision [§ 1.3.5], comme l'ont par exemple montré Bryant [Bry86] pour les OBDDs, Srinivasan et al. [SK$^+$90] pour les MDDs, Gergov et Meinel [GM94] pour les FBDDs ou encore Vempaty [Vem92] pour les automates à états finis.

Sur des langages-cibles bien choisis, cette méthode a l'avantage d'être rapide. Elle a en revanche un sérieux inconvénient : lors de la construction, les formes compilées temporaires peuvent être beaucoup plus grosses que la forme compilée finale. Si la mémoire disponible ne suffit pas à contenir toutes les structures intermédiaires, la compilation échoue, même si la forme compilée finale aurait eu une taille raisonnable.

Trace d'un solveur

Une autre approche consiste à exploiter les relations entre l'arbre de recherche de la résolution d'un problème et la représentation de ce problème dans un langage-cible de compilation. Cette idée est à l'origine de l'algorithme *« DPLL with a trace »* de Huang et Darwiche [HD04] (approfondie par Wille, Fey et Drechsler [WFD07] et Huang et Darwiche [HD05]). L'algorithme DPLL [DLL62] est une procédure fondamentale pour vérifier la cohérence d'une CNF. Grossièrement parlant, elle fonctionne de manière récursive, en sélectionnant une variable et en raisonnant par disjonction des cas sur sa valeur ; l'idée est que la formule est cohérente si et seulement si au moins l'un des cas résulte en une formule cohérente.

Huang et Darwiche [HD05] ont remarqué que l'arbre de recherche de l'algorithme DPLL (étendu pour énumérer tous les modèles au lieu de s'arrêter au premier rencontré) correspond exactement à un FBDD non réduit. Ils ont adapté cet algorithme pour qu'il puisse également compiler des OBDDs et des d-DNNFs. Cette technique ne se limite pas aux langages booléens ; tout solveur utilisant un algorithme de branchement peut être utilisé pour compiler des diagrammes de décision. Le principe a notamment été adapté à la compilation de MDDs [HH$^+$08], et a été

proposé par Wilson [Wil05] pour la compilation de SLDDs. Contrairement à la méthode *bottom-up*, cette construction *top-down* ne génère aucune structure intermédiaire qui soit plus grosse que le résultat final. Néanmoins, elle est généralement plus lente, puisqu'elle nécessite l'exploration de l'arbre de recherche entier et l'énumération de toutes les solutions. Cet inconvénient peut être limité par l'utilisation d'un *cache*, qui permet d'éviter d'explorer plusieurs fois des sous-problèmes équivalents.

1.5.3 Méthodes non-exactes de compilation

Compilation approchée

Un problème courant en pratique avec la compilation de connaissances est la taille importante de la forme compilée ; même en utilisant les langages les plus succincts supportant les opérations voulues, la mémoire disponible dans les systèmes embarqués peut ne pas suffire. O'Sullivan et Provan [OP06] ont donc proposé une méthode permettant de réduire la taille des formes compilées, en ne considérant que les solutions les plus intéressantes d'un problème.

L'idée consiste à associer une valuation à chaque solution, représentant sa vraisemblance (par exemple une probabilité : plus elle est proche de 1, plus la solution est intéressante), et à ne garder dans la forme compilée que les solutions dont la valeur est plus grande qu'un seuil donné. En définissant la valuation d'un ensemble de solutions comme étant la somme des valuations de chaque solution, il est possible de calculer la valuation v_c de la forme compilée entière φ_c (qui contient toutes les solutions possibles), et celle v_p de la compilation partielle φ_p (qui contient toutes les solutions dont la valuation est supérieure au seuil).

Le ratio $v_\mathrm{p}/v_\mathrm{c}$ est appelé *ratio de couverture de valuation*, et représente la proportion de « bonnes » solutions que couvre la forme compilée partielle. Le ratio $\|\varphi_\mathrm{p}\|/\|\varphi_\mathrm{c}\|$ est le *ratio de réduction de mémoire*, qui représente la proportion d'espace mémoire que la compilation partielle permet de gagner. Grâce à ces deux grandeurs, il est possible de déterminer la qualité d'une compilation approchée, et de choisir une valeur appropriée pour le seuil.

Compilation itérative

L'idée de la compilation approchée a été poussée plus loin par Venturini et Provan [VP08a], qui ont proposé des algorithmes (pour les langages `IP` et `DNNF`) permettant de construire des *solutions partielles*, c'est-à-dire des instanciations incomplètes des variables. Ces solutions sont alors itérativement complétées, tout en assurant que leur valuation reste toujours au-dessus d'un seuil donné. La structure compilée finale contient donc toutes, et seulement, les modèles les plus intéressants de la fonction initiale.

*
**

Dans ce chapitre, nous avons présenté la compilation de connaissances, une technique consistant à traduire un problème hors-ligne pour accélérer sa résolution en ligne. Nous avons choisi l'angle de la carte de compilation, en détaillant ses concepts par le biais d'un cadre très général. Nous avons présenté un certain nombre de langages booléens de la littérature en tant que sous-langages d'un langage-cible à portée générale que nous avons introduit, `GRDAG`. Inspiré par le cadre `VNNF`, il permet notamment de regrouper tous les langages de type « diagramme de décision », que leurs variables soient booléennes ou non. Nous avons ensuite rassemblé les résultats relatifs à la carte de compilation de ces langages. Enfin, nous avons présenté succinctement des méthodes pratiques de compilation vers ces langages-cibles. Nous pouvons à présent passer au chapitre suivant, qui traite du second domaine de l'IA sur lequel cette thèse porte, à savoir la planification automatisée.

CHAPITRE

2

Planification

Un système autonome doit être capable de *prendre des décisions* adéquates à sa situation et sa mission courante. La prise de décision implique un raisonnement à propos des alternatives possibles, de leurs résultats et de leur combinaison, de manière à pouvoir en choisir une qui soit un bon « premier pas » vers le succès de la mission. Par définition d'un système autonome, ce raisonnement ne peut être effectué par un opérateur humain. Une possibilité est de s'appuyer sur la *planification automatisée*.

Dans ce chapitre, nous présentons les grandes lignes de la planification automatisée, en commençant par donner une définition formelle générale du domaine [§ 2.1], puis en décrivant diverses orientations et techniques de la planification, que nous appelons *paradigmes de planification* [§ 2.2], et enfin en précisant comment la compilation de connaissances s'applique à la planification [§ 2.3].

2.1 Définition générale

2.1.1 Intuition

D'un point de vue général, la planification vise à répondre à la question suivante : « que faire pour atteindre tel objectif ? ». En ce qui concerne la planification *automatisée*, la réponse à cette question doit être un algorithme, appelé *planificateur*. Cette première définition intuitive est délibérément vague, en effet la planification automatisée peut prendre diverses formes ; nous allons expliquer en quoi notre définition est ambiguë.

Tout d'abord, la réponse à la question « que faire » dépend évidemment de ce qui *peut* être fait, c'est-à-dire des *actions possibles*. Cela soulève la question de la

représentation du monde ; dans le cadre de la planification, celui-ci est généralement modélisé par un *système états-transitions*, avec des actions et des événements modifiant l'état courant du monde.

Ensuite, la notion d'« objectif à atteindre » peut désigner diverses exigences. Il peut s'agir d'un but de mission fini et fixé (par exemple, la mission est accomplie si tel endroit est atteint) ou au contraire d'un objectif continu ou perpétuel (par exemple, suivre en permanence un chemin donné sans jamais s'arrêter). Dans ce dernier cas, le but n'est jamais *atteint* à proprement parler, mais doit être assuré en permanence ; les problèmes de planification ayant de tels objectifs sont appelés *problèmes de contrôle*. Il peut également y avoir plusieurs buts, d'importance variable pour la mission, auquel cas le planificateur doit *optimiser* sa réponse en fonction de certaines préférences.

Enfin, la définition est ambiguë en ce qui concerne la forme que doit avoir la réponse. Veut-on une suite d'actions ? A-t-on besoin que le choix des actions dépende de certaines conditions ? Jusqu'à quel point a-t-on besoin que la solution soit robuste aux incertitudes et aux aléas ?

Nous précisons toutes ces ambiguïtés dans la suite de cette section, en commençant par la question de la représentation du monde [§ 2.1.2], puis en détaillant les autres éléments d'un problème de planification [§ 2.1.3] ; enfin, nous présentons différentes catégories de solutions attendues [§ 2.1.4]. Cette section se veut être un survol du domaine, et ne vise pas l'exhaustivité. Le lecteur intéressé peut se référer au livre de référence de Ghallab, Nau et Traverso [GNT04], ou à de récents tutoriels tels ceux de Rintanen [Rin11] ou de Geffner [Gef11].

2.1.2 Description du monde

Lors de la définition d'un problème de planification, on ne prend évidemment pas en compte l'univers tout entier. Il est plus naturel de décrire un certain nombre d'éléments importants, avec lesquels l'*agent* (le système autonome en question dans le problème) peut interagir. Le raisonnement ne peut alors s'appliquer qu'à ces éléments — ils constituent une restriction du monde. Cette section présente la façon dont ces éléments sont généralement décrits, de manière à pouvoir faire l'objet de raisonnements.

Modèle de base

Le modèle général du monde est souvent un système états-transitions [GNT04, § 1.4].

Définition 2.1.1. Un *système états-transitions* est un quadruplet $\Sigma = \langle S, A, E, \gamma \rangle$, où

- S est un ensemble récursivement énumérable d'*états* ;
- A est un ensemble récursivement énumérable d'*actions* ;
- E est un ensemble récursivement énumérable d'*événements* ;

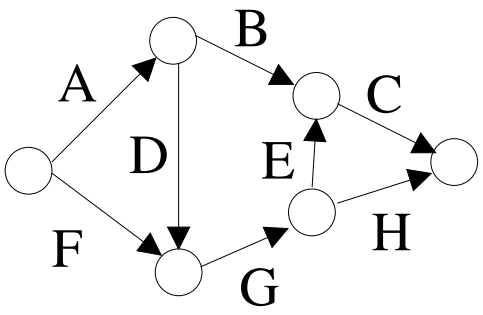


Fig. 2.1 : Éléments d'un problème de planification. Les cercles représentent des états, et les arcs (A, B, C…) représentent des actions, qui font passer le monde d'un état à un autre.

- γ: $S \times A \times E \to 2^S$ est une *fonction de transition d'états*.

Ce modèle permet de représenter la dynamique du monde. L'état courant peut être modifié par un événement, sur lequel l'agent n'a aucun contrôle, ou par une action, effectuée par l'agent. Les événements et les actions sont appelés des *transitions d'états*, d'où le nom du système. Un système états-transitions peut être vu comme un multigraphe, dont les nœuds sont les états du système, et les arcs sont les actions et les événements. Le diagramme en figure 2.1 est une représentation graphique d'un système états-transitions modélisant le monde pour un problème de planification très simple.

On n'utilise généralement pas comme modèle un système états-transitions dans cette forme basique. Il existe de nombreux paradigmes, chacun faisant des hypothèses restrictives sur le modèle ou l'améliorant en ajoutant de l'information à certains de ses éléments. Ces modifications visent soit à faciliter le raisonnement, soit à exprimer les aspects spécifiques de certains domaines d'application.

Restrictions usuelles du modèle

Les restrictions les plus courantes de la définition de base d'un système états-transitions sont les suivantes :

- ensembles *finis* d'états, d'actions et d'événements ;
- système *statique* — il n'existe aucune dynamique incontrôlable, E est un singleton ne contenant que l'événement « vide » ;
- système *déterministe*, dans lequel les effets des actions sont certains — appliquer une action donnée dans un état donné mène toujours au même état, c'est-à-dire, $\forall s \in S, \forall \langle a, e\rangle \in A \times E, |\gamma(s, a, e)| = 1$.

Ces hypothèses sont parfois prises toutes à la fois, comme c'est le cas dans la *planification classique* [§ 2.1.3.3], mais ce n'est pas systématique. Chaque paradigme de planification a son propre ensemble de restrictions.

Extensions usuelles du modèle

Même lorsqu'aucune de ces restrictions n'est appliquée, l'expressivité du modèle basique de la définition 2.1.1 reste limitée. Il arrive souvent que Les paradigmes de planification l'affinent, en ajoutant de l'information à tel ou tel élément. Ainsi,

- les états peuvent avoir divers niveaux d'*observabilité*, c'est-à-dire que l'état courant peut ne pas être entièrement connu par le système (cela peut notamment être le cas pour l'état initial) ;
- les transitions peuvent être *probabilistes*, ce qui hiérarchise les résultats possibles des actions [§ 2.2.4] ;
- les actions peuvent être non-instantanées, ce qui permet de prendre en compte le temps [§ 2.2.5.4].

Toutes ces modifications — restrictions et extensions — n'affectent que le modèle *théorique* du monde. En pratique, un planificateur manipule une *représentation* de ce modèle abstrait. Le choix de la représentation concrète à utiliser diffère également d'un paradigme à un autre.

Représentation du modèle

Il peut y avoir un nombre gigantesque d'états dans un monde. Ce nombre n'est en fait même pas nécessairement fini, dans le cas général. Pourtant, le planificateur doit pouvoir lire en entrée une description du monde ; il a donc besoin d'une *représentation implicite* de ce monde, dans laquelle les états ne sont pas énumérés *in extenso*.

Une représentation possible, basée sur la logique propositionnelle, exprime les états comme des *instanciations* de *variables d'état*, et les transitions comme des *formules* sur ces variables. Considérons par exemple deux variables d'état booléennes *light-on* et *door-open*. Leur instanciation décrit quatre états possibles :

- *light-on* $= \bot$, *door-open* $= \bot$: l'état dans lequel la lumière est éteinte et la porte est fermée ;
- *light-on* $= \bot$, *door-open* $= \top$: l'état dans lequel la lumière est éteinte et la porte est ouverte ;
- *light-on* $= \top$, *door-open* $= \bot$: l'état dans lequel la lumière est allumée et la porte est fermée ;
- *light-on* $= \top$, *door-open* $= \top$: l'état dans lequel la lumière est allumée et la porte est ouverte.

Les actions sont généralement définies par deux types de formules sur les variables d'état, les *préconditions* et les *effets*. Pour qu'une action donnée soit *exécutable* dans un état donné, les préconditions de cette action doivent être vérifiées

Fig. 2.2 : Un système états-transitions exprimé avec des fluents. Chaque carré contenant une lettre majuscule est une action, avec ses préconditions à gauche (les fluents qui doivent être vrais pour que l'action soit exécutable) et ses effets à droite (les fluents qui deviennent vrais ou faux une fois que l'action a été exécutée).

dans cet état. Une fois que l'action a été effectuée, le monde se retrouve dans un état dans lequel les effets sont vérifiés.

L'évolution du monde est souvent vue comme une suite d'étapes ; chaque étape correspond à un état donné, et donc à une certaine instanciation des variables d'état. Dans ce cas, les variables d'état sont appelées *fluents*, et les préconditions et effets sont usuellement représentés comme des ensembles de fluents : l'ensemble des fluents qui doivent être vrais (préconditions), l'ensemble des fluents qui deviennent vrais (effets positifs), et l'ensemble des fluents qui deviennent faux (effets négatifs). Un exemple de système états-transitions exprimé à l'aide de fluents est donné en figure 2.2.

2.1.3 Définition d'un problème de planification

Nous avons à présent une idée de la façon dont le monde est décrit dans un problème de planification. Cela n'est cependant pas suffisant pour complètement définir un problème ; sur un modèle donné du monde peuvent se baser un grand nombre de problèmes différents, selon l'état initial et l'objectif désirés.

États initiaux

Dans un système états-transitions donné, il est bien sûr possible de choisir comme état initial un état quelconque. Ainsi, si le monde modélisé décrit les chemins possibles d'un véhicule automatisé, on peut considérer tous les problèmes visant à atteindre une certaine position en partant de n'importe quelle position. Dans les problèmes de planification classique définis avec des fluents, l'état initial est spécifié par un ensemble de fluents — ceux qui sont vrais dans cet état-ci. Dans l'exemple de la figure 2.2, on pourrait établir l'état initial à $\{i\}$; cela signifierait qu'au départ, i est vrai et tous les autres fluents sont faux.

Il n'est pas obligatoire que l'état initial soit unique. Certains paradigmes, tels la planification conformante [SW98] ou la planification par *model-checking* [GT99] utilises des états initiaux multiples ; cela peut notamment permettre d'introduire de l'incertitude dans le problème sans utiliser d'actions non-déterministes.

La notion de but

Intuitivement, la planification vise à trouver un plan permettant d'atteindre un certain état ; mais les objectifs peuvent être beaucoup plus complexes. Les objectifs

« basiques » correspondent à ce qu'on appelle *objectifs d'atteignabilité* ; ils sont simplement définis par un état, ou un ensemble d'états. Dans la représentation par fluents, cet ensemble d'états peut être exprimé comme un ensemble de fluents — ceux dont on veut qu'ils soient vrais au final.

Il est possible de définir des types d'objectifs plus complexes. Avec les objectifs d'atteignabilité, un plan est une solution ou ne l'est pas ; il n'y a pas d'autre cas à envisager. Considérons au contraire un exemple où un robot explorateur doit ramasser autant d'échantillons que possible. Cet objectif n'est pas « atteignable », mais induit une classification des déroulements possibles ; la solution est donc un plan permettant d'obtenir le *déroulement préféré*.

D'autres exemples ne font même pas intervenir d'évaluation des déroulements. Supposons que le robot explorateur doive ramasser des échantillons tout en restant à portée du contrôle radio, pour pouvoir revenir à la station dès que nécessaire. L'objectif ici est une *condition* qui doit être remplie durant toute l'exécution de la mission. Les objectifs de ce type sont appelés *objectifs étendus* [DLPT02, PT01].

Problèmes de planification

Les *problèmes de planification* sont définis grâce aux divers éléments que nous venons de présenter : un problème de planification générique est défini par un système états-transitions, un ensemble d'états initiaux et un objectif. On classifie les problèmes selon le type et les propriétés de leurs éléments. Par exemple, un problème de *planification classique* est défini comme suit [FN71].

Définition 2.1.2. Un *problème de planification classique* est défini par un triplet $P = \langle \Sigma, s_0, G \rangle$, où $\Sigma = \langle S, A, \gamma \rangle$ est un système états-transitions fini, statique et déterministe, $s_0 \in S$ est l'état initial, et $G \subseteq S$ est l'ensemble des états-buts.

Relâcher une des restrictions du système états-transitions, l'étendre, ou utiliser un objectif étendu sont autant de manières d'obtenir différents types de problèmes de planification. Mis à part la planification classique, les catégories de problèmes n'ont généralement pas de nom spécifique ; elles sont souvent nommées d'après le paradigme qui les utilise, comme c'est le cas des « problèmes de planification par MDP » [voir section 2.2.4], ou simplement décrits extensivement, comme les « problèmes de planification non-déterministe sous observabilité partielle avec objectifs d'atteignabilité ».

De nombreux *langages de description* ont été créés, dans le but d'unifier les représentations de problèmes de planification et de rendre compte, de manière claire et standardisée, des différentes capacités de chaque algorithme de planification. Le premier à avoir été largement utilisé était le langage STRIPS, créé comme le langage d'entrée du *Stanford Research Institute problem solver* [FN71] et ne visant donc pas à l'exhaustivité, mais seulement à couvrir les possibilités offertes par ce solveur.

Ce langage a été étendu par ADL, *action description language*, qui permet notamment d'exprimer des disjonctions de fluents dans les préconditions et les effets. Le standard actuel est PDDL, *planning domain definition language*, version 3.1

[voir Kov11 pour une spécification complète]. Ce dernier est plus expressif que STRIPS et PDDL, permettant notamment d'utiliser des durées pour les actions, des fluents numériques, des coûts d'action, des préférences sur les états, et des prédicats dérivés. Une extension de PDDL, PDDL+, permet par surcroît de manipuler conjointement des variables discrètes et continues [FL06].

2.1.4 Solutions d'un problème de planification

Comme nous l'avons vu, il existe de multiples sortes de problèmes de planification. Nous allons à présent définir ce qu'est une *solution* d'un problème de planification.

Forme des solutions

Les solutions possibles peuvent prendre diverses *formes*. La plus basique est simplement une suite d'actions, que l'on appelle un *plan*.

Définition 2.1.3 (Plan)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions. Un *plan* π poun Σ est une suite finie d'actions de A : $\pi = \langle a_1, \ldots, a_n \rangle \in A^n$, avec $n \in \mathbb{N}$ l'*horizon* de π.

Tous les plans ne sont pas des solutions, ils ne sont même pas tous faisables — nous formaliserons cela dans les deux sous-sections suivantes. Intuitivement, un plan est solution d'un problème s'il « permet d'atteindre le but » quand les actions sont exécutées dans le bon ordre en partant de l'état initial. Si nous prenons, par exemple, un état initial défini par $\{i\}$ dans le problème de planification classique de la figure 2.2, un plan-solution du problème consistant à atteindre un état où f est vrai est $\langle B, A, C \rangle$.

Il y a diverses sortes de plans : ils peuvent être notamment soit *séquentiels*, soit *parallèles* (si les actions peuvent être exécutées simultanément). Quand les actions à exécuter dépendent de certaines observations, les plans peuvent être *conditionnels*.

Outre les plans, les *politiques* constituent une autre forme de solution.

Définition 2.1.4 (Politique de décision)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions. Une *politique de décision* δ pour Σ est une fonction $\delta \colon S \to A$.

La politique δ peut être *partielle*, c'est-à-dire non définie sur l'ensemble entier des états.

Contrairement aux plans-solutions, qui fournissent une suite ordonnée d'actions à exécuter, une politique est une *fonction* allant de l'ensemble des états à celui des actions. Elle indique la prochaine décision à prendre en fonction de l'état courant. Une fois que cette décision a été exécutée, le système doit interroger la politique à nouveau pour déterminer l'action suivante, et ainsi de suite.

Les politiques sont plus robustes aux incertitudes que les plans : au cours de l'exécution, si l'une des décisions n'a pas eu l'effet escompté, un plan peut devenir inutile (si les décisions restantes ne sont pas adaptées à l'état inattendu). Avec une politique, même si l'on se retrouve dans un état inattendu, il est possible d'avoir à disposition l'action à effectuer.

La forme attendue de la solution à un problème dépend donc grandement du modèle : les politiques sont bien adaptées aux problèmes non-déterministes, mais nécessitent une modification du modèle pour être capables d'exprimer une suite fixe d'actions (il faut en effet ajouter des variables d'état spéciales pour se souvenir de l'« historique » des actions). Les plans conditionnels sont adaptés aux problèmes partiellement observables, mais ne permettent pas à une action d'être répétée indéfiniment tant qu'un état donné n'est pas atteint.

Faisabilité des solutions

Pour définir ce que sont un *plan solution* et une *politique solution*, nous devons pouvoir exprimer le fait que les plans et politiques ne sont pas systématiquement *exécutables*, ou *faisables*. La notion de faisabilité signifie intuitivement que le plan ou la politique est compatible avec les transitions du modèle. La définition formelle d'un plan faisable [adaptée de FG00] s'appuie sur la notion d'*historique*.

Définition 2.1.5 (Historique)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions, et $\pi = \langle a_1, \ldots, a_n \rangle$ un plan pour Σ. Un *historique pour* π est une suite d'états $\langle h_0, \ldots, h_k \rangle \in S^k$, avec $0 \leqslant k \leqslant n$, qui vérifie

$$\forall i \in \{1, \ldots, k\}, h_i \in \gamma(h_{i-1}, a_i).$$

L'état h_0 est l'*état de départ* de cet historique, h_k son *état d'arrivée*, et k sa *taille*.

Un historique est donc une suite d'états compatibles avec le plan π : l'application de ce plan à partir de l'état de départ *peut* mener à l'état d'arrivée.

Définition 2.1.6 (Plan faisable)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions, et $s_0 \in S$. Un plan $\pi = \langle a_1, \ldots, a_n \rangle$ pour Σ est dit *faisable depuis* s_0 (ou *(toujours) exécutable depuis* s_0) si et seulement si tout historique pour π de taille $k < n$ et débutant par s_0 vérifie $|\gamma(h_k, a_{k+1})| > 0$.

Un plan est faisable depuis un état initial donné si la première action est applicable dans l'état initial, la seconde applicable dans tous les états résultants possibles, etc. Pour qu'un plan soit faisable, il ne doit exister aucun historique menant à un état dans lequel l'action courante n'est pas applicable.

Pour définir les historiques des politiques de décision, il nous faut introduire le concept de *structure d'exécution*.

Définition 2.1.7 (Structure d'exécution)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions, et δ une politique. La *structure d'exécution* induite par δ sur Σ est l'ensemble $\Sigma_\delta = \{ \langle s, s' \rangle \in S \times S \mid s' = \gamma(s, \delta(s)) \}$.

Une structure d'exécution est donc un système états-transitions simplifié, dans lequel il ne reste aucune action ; on peut la voir comme un graphe orienté, où les arcs sont les transitions compatibles avec la politique. Nous pouvons maintenant définir les historiques pour les politiques de décision.

Définition 2.1.8 (Historique)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions, $s_0 \in S$, et δ une politique. Un *historique de δ depuis s_0* est un chemin complet dans la structure d'exécution Σ_δ débutant par l'état s_0.

À noter que contrairement aux historiques pour les plans, les historiques pour les politiques peuvent être infinis. Ils sont cependant par définition des *chemins complets*, c'est-à-dire qu'ils ne peuvent s'arrêter que dans des états pour lesquels la politique ne fournit aucune action à effectuer. La notion d'historique va nous être utile dans la sous-section suivante ; introduisons pour le moment la notion de *politique exécutable* [GT99].

Définition 2.1.9 (Politique exécutable)**.** Soit $\Sigma = \langle S, A, \gamma \rangle$ un système états-transitions. Une politique δ pour Σ est dite *exécutable* si et seulement pour tout couple $\langle s, a \rangle$ tel que $a = \delta(s)$, il existe un état $s' \in S$ vérifiant $s' = \gamma(s, a)$.

Une politique est exécutable si toute action qu'elle renvoie est exécutable dans l'état courant. Remarquons que cette définition est plus simple que celle de plan faisable.

Force des solutions

Pour constituer une solution d'un problème de planification, un plan ou une politique doit nécessairement être exécutable. Cela n'est néanmoins pas suffisant ; la deuxième condition est qu'il ou elle doit permettre d'atteindre l'objectif.

Définition 2.1.10 (Solution en planification classique)**.** Soit $P = \langle \Sigma, s_0, G \rangle$ un problème de planification classique. Un plan π pour Σ est une *solution* de P si et seulement s'il est faisable depuis s_0 et s'il existe un historique pour π depuis s_0 qui arrive dans un des états-buts $g \in G$.

Pour un problème non classique, la définition d'une solution n'est en fait pas unique ; en effet, le problème peut être plus ou moins sujet à de l'*incertitude*, par le biais du non-déterminisme ou de l'observabilité partielle par exemple. Les exigences quant à la robustesse de la solution peuvent alors varier. Pour certaines applications, l'objectif doit être atteint dans tous les cas, malgré le non-déterminisme : on parle alors de solution *forte*. Pour d'autres applications, cette robustesse n'est pas strictement nécessaire ; on peut se permettre d'être optimiste et de ne chercher qu'une solution dite *faible*.

La force d'un plan-solution est liée à la notion d'*historique*, définie dans la sous-section précédente. On peut définir divers types de solution pour chaque catégorie de problème de planification ; nous allons donner quelques exemples, qui nous seront particulièrement utiles dans la suite. Les solutions les plus simples sont les solutions *faibles*, qui ne requièrent que l'*existence* d'un historique [GT99].

Définition 2.1.11 (Solution faible)**.** Soit $P = \langle \Sigma, S_0, G \rangle$ un problème de planification avec plusieurs états initiaux et objectifs d'atteignabilité. Un plan π (resp. une politique δ) pour Σ est une *solution faible* de P si et seulement si pour tout $s_0 \in S_0$, il (resp. elle) est exécutable depuis s_0, et il existe un historique pour π (resp. δ) depuis s_0 arrivant dans un des états-buts $g \in G$.

Pour qu'une politique soit une solution faible, il doit y avoir, dans la structure d'exécution, au moins un chemin depuis chaque état initial vers l'un des états-buts. Pour qu'elle soit forte, il ne doit en revanche exister *aucun* chemin ne menant pas à un état-but [CRT98b].

Définition 2.1.12 (Solution forte)**.** Soit $P = \langle \Sigma, S_0, G \rangle$ un problème de planification avec plusieurs états initiaux et objectifs d'atteignabilité. Une politique δ pour Σ est une *solution forte* de P si et seulement si elle est exécutable et si quel que soit $s_0 \in S_0$, tous les historiques pour δ depuis s_0 sont finis et arrivent dans un des états-buts $g \in G$.

Quand il s'agit d'un plan, une solution forte est appelée *conformante*.

Définition 2.1.13 (Solution conformante)**.** Soit $P = \langle \Sigma, S_0, G \rangle$ un problème de planification avec plusieurs états initiaux et objectifs d'atteignabilité. Un plan π pour Σ, d'horizon n, est une *solution conformante* de P si et seulement si quel que soit $s_0 \in S_0$, il est faisable depuis s_0, et tous les historiques pour π de taille n et débutant par s_0 arrivent dans un des états-buts $g \in G$.

On cherche généralement des solutions conformantes pour des problèmes entièrement non-observables, pour lesquels le plan doit être valide alors même que le système n'a aucune information sur son état courant [voir e.g. SW98]. Si le problème est (partiellement) observable, ce type de solution est plutôt appelé *contingente* [DHW94]. Pour finir, nous définissons un moyen terme entre les solutions faibles et fortes [CRT98a].

Définition 2.1.14 (Solutions cycliques fortes)**.** Soit $P = \langle \Sigma, S_0, G \rangle$ un problème de planification avec plusieurs états initiaux et objectifs d'atteignabilité. Une politique δ pour Σ est une *solution cyclique forte* de P is et seulement si elle est exécutable, et si quel que soit $s_0 \in S_0$, tous les historiques finis pour δ depuis s_0 arrivent dans un des états-buts $g \in G$.

Insistons sur la différence avec les solutions fortes : il peut exister des historiques infinis, mais ils ne sont pas pris en compte. La seule exigence est que les historiques finis ne doivent pas arriver dans des états non buts. L'idée est que l'on considère que les boucles infinies sont très improbables — une solution cyclique forte permet donc d'atteindre l'objectif de manière « presque certaine ».

2.2 Paradigmes de planification

De nombreuses techniques ont été utilisées, au cours des dernières décennies, pour résoudre les problèmes de planification. Certains techniques visent à résoudre des catégories de problèmes spécifiques ; d'autres cherchent à être aussi génériques que possible. Dans cette section, nous présentons plusieurs paradigmes de planification particulièrement liés à notre sujet.

2.2.1 Planification en avant dans l'espace des états

Le paradigme de planification le plus simple consiste à raisonner directement sur le système états-transitions, c'est-à-dire à utiliser une *recherche dans l'espace des états* [voir e.g. GNT04, chap. 4]. L'idée est d'essayer d'atteindre l'objectif, en partant de l'état initial ; en d'autres termes, de trouver un *chemin* de l'état initial à un état-but dans le graphe représentant le système. S'il n'existe aucun chemin de ce type, le problème n'a pas de solution. S'il en existe un, le plan correspondant à la suite d'actions constituant le chemin est une solution.

Algorithme 2.1 Recherche en avant dans l'espace des états.

fonction : `ForwardSS`(P, s)
entrée : un problème de planification classique $P = \langle \Sigma, s_0, S_G \rangle$
entrée : un état courant s
sortie : un plan π de s à un but s'il en existe un, **nil** sinon
si $s \in S_G$ **alors**
    renvoyer le plan vide
soit A_s l'ensemble des actions a tel que $|\gamma(s, a)| \neq 0$
tant que $A_s \neq \varnothing$ **faire**
    prendre une action a dans A_s (et la retirer)
    soit $s' \coloneqq \gamma(s, a)$
    soit $\pi \coloneqq$ `ForwardSS`(P, s)
    si $\pi \neq$ **nil alors**
        renvoyer $a \,.\, \pi$
renvoyer nil

L'algorithme 2.1 présente cette procédure de manière récursive. L'approche peut également être utilisée en arrière — en partant des buts et en tentant d'atteindre l'état initial. Cette méthode assez naturelle a pour défaut que l'espace de recherche est généralement gigantesque : au pire cas, *toutes les suites d'actions possibles* sont explorées. C'est pourquoi ce paradigme n'est en pratique jamais utilisé tel quel, mais toujours combiné avec des *heuristiques* pour le choix de l'action suivante [voir section 2.2.5.2], ou avec des restrictions quant aux actions possibles, comme dans l'algorithme STRIPS [FN71].

2.2.2 Planification par satisfiabilité

L'idée derrière le paradigme de *planification par satisfiabilité* [KS92b] est de résoudre les problèmes de planification en utilisant des solveurs SAT (des programmes pouvant décider de la cohérence d'une formule propositionnelle donnée). En effet, ces solveurs faisant l'objet de recherches poussées depuis de nombreuses années, des algorithmes efficaces ont été développés, et il semble intéressant de tenter de tirer parti de leurs performances pour la planification.

Pour ce faire, on « encode » un problème de planification $P = \langle \Sigma, s_0, g \rangle$ en une formule propositionnelle φ, de façon à ce que tout modèle de φ corresponde à un plan-solution de P. En général, on décompose l'horizon en *étapes*, chaque étape correspondant à l'application d'une certaine action dans un certain état du monde. Il est donc nécessaire que chaque étape dispose de ses propres variables d'état et d'action, dont l'instanciation indique l'état courant et l'action exécutée à cette étape. Évidemment, cela empêche d'avoir un horizon non borné — qui nécessiterait un nombre infini d'états et d'actions, et donc une formule propositionnelle infinie.

Le problème de planification est par conséquent restreint : on cherche un plan *d'horizon* n (avec n un entier fixé). Le nombre d'étapes est alors fini ; on sait que chaque variable d'action a sera dupliquée en n versions $a_0, a_1, \ldots, a_{n-1}$, et chaque variable d'état s en $n+1$ versions $s_0, s_1, \ldots, s_n$. L'encodage du problème est à présent naturel ; nous allons l'illustrer sur un problème de planification classique P. Dans ce cas, l'encodage φ_P^n est tout simplement la conjonction des formules représentant les contraintes suivantes :

- l'instanciation des variables d'état à l'étape 0 doit représenter l'état initial ;
- l'instanciation des variables d'état à l'étape n doit représenter l'état-but ;
- quel que soit i entre 0 et $n-1$, en notant a_i l'action representée par l'instanciation des variables d'action à l'étape i, les variables d'état à l'étape i doivent respecter les préconditions de a_i, et les variables d'état à l'étape $i+1$ doivent respecter ses effets.

Ce n'est pas tout ; cet encodage n'empêche pas les variables d'état non modifiées par l'action courante de changer, ce qui est censé être interdit par l'*axiome du cadre* — et il n'empêche même pas l'exécution de plusieurs actions lors d'une même étape. Un encodage correct doit bien sûr inclure des formules interdisant de tels comportements.

La formule finale φ_P^n est équivalente au problème initial de planification bornée, au sens où chaque modèle de la formule est une solution du problème. On peut en effet extraire un plan d'un modèle $\vec{x}$: c'est simplement la suite d'actions $\langle a_0, \ldots, a_{n-1} \rangle$ — nous noterons ce plan $\pi_{\vec{x}}$. Par définition, ce plan est solution du problème de planification, et la réciproque est également vraie, comme le résume la proposition suivante [KS92b].

Proposition 2.2.1. Soit P un problème de planification classique, et soit φ_P^n son encodage propositionnel (tel que décrit précédemment). Soit $\vec{x}$ une Scope(φ_P^n)-instanciation ; il est démontré que $\vec{x} \in \text{Mod}(\varphi_P^n)$ si et seulement si $\pi_{\vec{x}}$ est un plan-solution de P en n étapes.

Pour trouver un plan-solution grâce à cette propriété, il suffit de lancer un solveur SAT sur φ_P^n, en augmentant progressivement la valeur de n (c'est-à-dire lancer le solveur pour $n = 1$, et s'il ne trouve pas de solution, incrémenter n et recommencer). Ce paradigme peut être affiné de diverses façons, en faisant varier par exemple le type de solveur utilisé, l'encodage des actions, ou l'encodage de l'axiome du

cadre [KMS96, GMS98]. Cette approche a notamment été étendue à la planification conformante [FG00, CGT03].

2.2.3 Planification par *model-checking*

La *planification par model-checking* [CG$^+$97, GT99] est un paradigme visant à résoudre des problèmes de planification non-déterministe. Il est basé sur le fait qu'une structure d'exécution, c'est-à-dire un système états-transitions « restreint » par une certaine politique [déf. 2.1.7], correspond exactement à une structure de Kripke [voir e.g. CGP99], à condition que les états soient encodés grâce à des variables propositionnelles. Pour que la politique soit une solution faible du problème de planification, il suffit qu'il existe un chemin de l'état initial à l'état-but. Pour que la politique soit une solution forte, tous les chemins partant de l'état initial doivent mener à l'état-but. Il est possible d'utiliser une formule CTL [Eme90] pour exprimer le fait que la politique est solution ; vérifier que telle politique est solution revient donc à effectuer du *model-checking* sur la structure d'exécution.

Cela permet de tirer parti des techniques de *model-checking* pour résoudre un problème de planification ; mais ce n'est pas le seul atout de cette approche, qui permet en sus d'appliquer à la planification toute l'expressivité de CTL — pour décider de la robustesse de la solution [CG$^+$97 pour la planification faible, CRT98b pour la planification forte, CRT98a pour la planification cyclique forte], mais aussi pour lui imposer des conditions continues, comme « garder en permanence le contact radio » [PT01].

Illustrons cette approche en nous concentrant sur la planification forte.

Définition 2.2.2 (Structure d'exécution en tant que structure de Kripke)**.** Soit $P = \langle \Sigma, S_0, S_g \rangle$ un problème de planification, et δ une politique sur $\Sigma = \langle S, A, \gamma \rangle$. La structure de Kripke correspondant à la structure d'exécution Σ_δ, notée $K_{P,\delta}$, a pour ensemble de mondes S, pour ensemble de mondes initiaux S_0, pour relation de transition l'ensemble $T = \{ \langle w, w' \rangle \in W^2 \mid w' \in \gamma(w, \delta(w)) \}$, et pour fonction d'étiquetage la fonction L associant à tout monde (ou état) w l'instanciation $\vec{s}$ des variables d'état qui représente w.

Comme dit précédemment, le fait que δ soit une solution forte du problème de planification P est équivalent au fait que $K_{P,\delta}$ vérifie une certaine propriété, qu'exprime la formule CTL $\mathbf{AF}g$, où g est la formule sur les variables d'état qui est vérifiée par les états de S_g, et **AF** est l'opérateur CTL signifiant « tous les chemins mènent à un monde où la formule est vraie » [GT99].

Proposition 2.2.3. En utilisant les notations précédentes, δ est une solution forte de P si et seulement si $K_{P,\delta} \models \mathbf{AF}g$.

Cette propriété permet de s'appuyer sur un algorithme de *model-checking* pour construire des politiques respectant les exigences.

2.2.4 Planification par processus décisionnels de Markov

Avec le système états-transitions standard de la définition 2.1.1, il n'y a aucun moyen de spécifier qu'un des résultats possibles de l'exécution d'une action non-déterministe est plus probable qu'un autre. Utiliser à la place un processus décisionnel de Markov [Bel57] permet de quantifier le non-déterminisme, par le biais de probabilités.

Définition 2.2.4 (Processus décisionnel de Markov)**.** Un *processus décisionnel de Markov* (MDP, *Markov decision process*) est un quadruplet $\Sigma = \langle S, A, P, R \rangle$, où :

- S est un ensemble fini d'états ;
- A est un ensemble fini d'actions ;
- $P \colon S \times A \times S \to [0, 1]$ est une distribution de probabilités sur les transitions d'état. $P(s, a, s')$ est généralement noté $P_a(s, s')$, et représente la probabilité d'atteindre s' lors de l'exécution de l'action a dans l'état s. Elle doit vérifier

  $$\forall s \in S, \forall a \in A, \sum_{s' \in S} P(s, a, s') \in \{0, 1\}$$

  (la somme vaut 0 si a n'est pas exécutable dans s, et 1 sinon) ;
- $R \colon S \times A \times S \to \mathbb{R}$ est la fonction de récompense. $R(s, a, s')$, souvent notée $R_a(s, s')$, est la récompense immédiate que gagne l'agent atteignant l'état s' après avoir exécuté l'action a dans l'état s.

À cause du non-déterminisme, dans un MDP donné, une politique δ qui associe une *unique* action à *chaque* état peut donner lieu à plusieurs déroulements possibles. Grâce aux probabilités sur les transitions, il est possible de quantifier la vraisemblance de chaque déroulement. On appelle le résultat de l'application d'une politique sur un MDP[1] un *historique*, comme dans le cas des transitions non-quantifiées [définition 2.1.8]. Un historique consiste en une suite infinie d'états, comme $h = \langle s_1, s_4, s_6, s_6, s_1, \ldots \rangle$. En notant $h = \langle h_i \rangle_{i \in \mathbb{N}}$, la probabilité d'obtenir h en appliquant δ est donnée par $P(h \mid \delta) = \prod_{i \in \mathbb{N}} P_{\delta(h_i)}(h_i, h_{i+1})$.

La fonction de récompense peut être utilisée comme un « objectif » pour un MDP. Elle permet de classer les historiques selon leur désirabilité ; on peut en effet définir des *fonctions d'utilité* sur les historiques grâce à elle, comme $V(h \mid \delta) = \sum_{i \in \mathbb{N}} \gamma^i R_{\delta(h_i)}(h_i, h_{i+1})$, avec γ un *facteur d'actualisation* (c'est-à-dire un paramètre dans $[0, 1[$, qui assure que l'utilité des historiques infinis est finie — plus précisément, il rend les récompenses proches plus importantes que les récompenses lointaines). Ainsi, en pratique, la fonction de récompense peut par exemple indiquer les états devant être évités (en leur associant une récompense négative) ou les états devant obligatoirement être visités (en leur associant une très forte récompense).

[1] Associé à une politique, un MDP devient un objet plus simple, appelé *chaîne de Markov*. Il s'agit en fait d'une structure d'exécution [définition 2.1.7] muni de transitions probabilistes.

La réunion de ces deux propriétés d'un historique permet de calculer l'*utilité espérée* d'une politique, en sommant les utilités de tous les historiques possibles, pondérés par leur probabilité : $E(\delta) = \sum_{h \in S^{\mathbb{N}}} P(h \mid \delta) \cdot V(h \mid \delta)$, avec $S^{\mathbb{N}}$ l'ensemble de tous les historiques possibles.

Cela nous permet finalement de définir ce qu'est un problème de planification par MDP.

Définition 2.2.5 (Problème de planification par MDP)**.** Un *problème de planification par MDP* est défini comme un MDP $\Sigma = \langle S, A, P, R \rangle$. Une *solution* d'un tel problème est une politique δ^* vérifiant $E(\delta^*) = \max_{\delta \in A^S} E(\delta)$.

En d'autres termes, résoudre un problème de planification exprimé sous la forme d'un MDP consiste à trouver une politique *optimale*, qui maximise l'utilité espérée.

2.2.5 Autres paradigmes

Il existe de nombreux autres paradigmes de planification. Nous en présentons rapidement quelques-uns dans cette section ; le lecteur intéressé peut se référer à l'exhaustif manuel de Ghallab, Nau et Traverso [GNT04].

Planification dans l'espace des plans partiels

La *planification dans l'espace des plans* [Sac75] est un paradigme de planification classique qui consiste à chercher dans l'espace des plans partiels, à la place de l'espace des états. Il n'y a pas d'*état courant* ; le raisonnement se situe au niveau global, sur l'horizon complet. La recherche s'effectue en identifiant les actions qui doivent apparaître dans le plan-solution, et en essayant d'ajouter des *contraintes d'ordonnancement*. Le retour en arrière est guidé par la connaissance des dépendances entre les actions, à l'aide de *liens causaux* et de *liaisons de variables*.

Planification par recherche heuristique

L'idée qui sous-tend la *planification par recherche heuristique* [BG01, HG00, HN01] est d'utiliser des heuristiques pour guider une recherche dans l'espace des états. Les heuristiques classiques nécessitent de calculer une certaine *distance à l'objectif* ; comme il s'agit généralement d'un problème difficile, la distance n'est pas calculée sur le problème réel, mais plutôt sur une *relaxation* — ignorant les effets négatifs des actions, par exemple.

Planification par réseau de tâches hiérarchisées

La planification par réseau de tâches hiérarchisées (HTN, *hierarchical task network*) [EHN96] est un paradigme qui vise à résoudre des problèmes différents de ceux que nous avons présentés jusqu'ici ; en effet, les objectifs n'y sont pas définis en tant qu'états, conditions ou fonctions de récompense, mais en tant que *tâches* à réaliser. Un problème HTN consiste en un problème de planification classique, muni d'un ensemble de *méthodes*, qui sont des « recettes » expliquant comment les tâches se décomposent en sous-tâches plus simples, elles-mêmes décomposées

en sous-tâches, etc. Les tâches les plus simples, indivisibles, sont appelées *tâches primitives*, et correspondent à des actions. L'intérêt de ce paradigme est que la définition de méthodes permet d'écarter certaines suites d'actions non pertinentes ; il s'agit d'une façon d'introduire de l'expertise humaine dans le modèle de planification.

Planification temporelle

Le but de la *planification temporelle* est de prendre en compte la durée des actions, et en particulier de leur définir des conditions et des effets plus complexes, comme des conditions devant être vérifiées tout au long de l'exécution, ou des effets n'étant vérifiés qu'à un instant spécifique. La planification temporelle est un domaine de recherche très vaste, comprenant de nombreux cadres différents ; citons notamment les travaux de Cushing et al. [CK+07], qui présentent une intéressante hiérarchie des sous-langages de PDDL selon leurs capacités temporelles.

Ordonnancement de ressources

L'*ordonnancement* est une branche de l'intelligence artificielle qui étudie les problèmes d'allocation de ressources et de temps pour un certain nombre de tâches. Pour qu'une tâche soit effectuée, il faut en général qu'un ensemble fixé d'*activités* soient traitées ; chaque activité utilise certaines ressources et prend un certain temps à être exécutée. L'objectif est alors de trouver un *ordonnancement* des activités, c'est-à-dire un plan indiquant *quand* chaque activité doit être exécutée et *quelles ressources* doivent lui être allouées ; cet ordonnancement doit respecter certaines contraintes, telles le non-partage d'une même ressource entre deux activités, le non-dépassement d'une durée fixée, etc. L'ordonnancement et la planification sont souvent considérés séparément, mais les deux domaines ont tendance à converger, car beaucoup de problèmes pratiques ne relèvent ni du pur ordonnancement ni de la pure planification [voir e.g. SFJ00].

2.3 Compilation pour la planification

Après ce survol de la planification automatisée, nous étudions comment certains des paradigmes que nous avons présentés tirent parti de la compilation de connaissances. On note $\mathcal{S}$ et $\mathcal{A}$ les ensembles de variables d'état et d'action, respectivement.

2.3.1 Planification par satisfiabilité

Planification classique

Barrett [Bar03] a traité un problème de planification déterministe et totalement observable, pour des systèmes embarqués temps réel avec des objectifs variables.

Il a proposé une approche de type planification par satisfiabilité en ligne. Le plan-solution n'est pas calculé hors ligne une fois pour toutes ; au contraire, pour chaque nouvel état rencontré, un nouveau plan est déterminé en ligne, ce qui rend le système plus robuste aux aléas.

Bien sûr, résoudre un problème SAT est difficile dans le cas général ; l'auteur a appliqué la compilation de connaissances, en traduisant son problème vers $\mathtt{DNNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ [§ 1.3.4]. Le travail restant à effectuer en ligne est le conditionnement du problème compilé suivant les observations courantes, l'oubli des états intermédiaires, et l'extraction d'un modèle pour récupérer un plan-solution. L'algorithme 2.2 présente cette procédure ; nous l'avons adapté de l'article originel pour que les requêtes et transformations utilisées apparaissent plus clairement.

Algorithme 2.2 La procédure en ligne utilisée par Barrett [Bar03].

1 : **entrée :** une DNNF booléenne φ représentant un problème de planification $\langle\langle S, A, \gamma\rangle, S, S_G\rangle$ (sans état initial spécifié) sur un horizon n
2 : **entrée :** une $\mathcal{S}_0$-instanciation $\overrightarrow{s_0}$ des variables d'état à l'étape 0
3 : **sortie :** un plan-solution
4 : calculer une DNNF φ' représentant $[\![\varphi]\!]_{|\overrightarrow{s_0}}$ *// conditionner φ par $\overrightarrow{s_0}$*
5 : calculer une DNNF φ'' représentant $\exists\mathcal{S}.[\![\varphi']\!]$ *// oublier états intermédiaires*
6 : **renvoyer** un modèle de φ''

Un des inconvénients de cette approche est que le plan est toujours de longueur n, étant donné que le problème de planification est codé avec un horizon fixé n. On peut contourner cette limitation en sélectionnant en priorité les modèles dans lesquels les actions sont effectuées tôt.

Planification conformante

Palacios et al. [PB$^+$05] ont étudié la possibilité d'utiliser le paradigme de planification par satisfiabilité pour résoudre des problèmes de planification conformante. Plus précisément, ils ont cherché à obtenir des plans parallèles conformants pour un problème de planification avec des actions non-déterministes et un état initial incertain. Dans le cadre classique de la planification par SAT, chaque modèle de la formule codant le problème est une solution ; ce n'est pas le cas ici, puisque le plan doit mener au but *quel que soit l'état initial*. Les procédures usuelles de résolution de SAT ne permettent pas de s'assurer que la solution vérifie cette contrainte. Ils fonctionnent en effet comme suit : une fois qu'une valeur a été choisie pour une variable, les instanciations incohérentes des autres variables sont rejetées. Cependant, il n'est pas garanti que *toutes* les instanciations restantes correspondent à des solutions fortes.

Pour trouver ces plans conformants, les auteurs ont proposé d'utiliser une étape d'élagage spécifique, basée sur la *validité* selon les états initiaux. Une instanciation partielle des variables d'action correspond en effet à un plan partiel ; si ce plan partiel ne couvre pas tous les états initiaux, il est inutile d'essayer d'instancier les variables restantes — le plan partiel est dit invalide, et peut être rejeté. Cependant,

tester si un plan partiel est valide selon les états initiaux est un problème difficile en général ; les auteurs ont donc identifié un langage-cible permettant cette vérification en temps polynomial.

La vérification en question revient à une projection existentielle de l'instanciation courante sur les variables d'état initial, suivie d'un comptage de modèles : en effet, si le nombre de modèles est inférieur au nombre d'états initiaux, cela signifie que certains états initiaux ne sont pas couverts. Les opérations nécessaires sur les formes compilées sont donc **FO** et **CT** ; les auteurs ont proposé de compiler le problème de planification dans le langage $\mathtt{d\text{-}DNNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$, en utilisant un arbre de décomposition particulier assurant que l'opération d'oubli maintient le déterminisme sur $\mathtt{DNNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$. La procédure est décrite dans l'algorithme 2.3.

Algorithme 2.3 Le planificateur conformant de Palacios et al. [PB$^+$05].

entrée : une d-DNNF booléenne φ représentant un problème de planification $\big\langle\langle S, A, \gamma\rangle, S_0, S_G\big\rangle$ sur un horizon n
sortie : un plan conformant
tant que il reste une variable d'action non déterminée **faire**
    sélectionner une variable d'action a_i *// voir l'article original pour des détails sur la sélection*
    calculer une d-DNNF φ' représentant $[\![\varphi]\!]_{|a_i=\top}$ *// assigner la variable d'action à* $\top$
    si le nombre de modèles de $\exists\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}.[\![\varphi']\!]$ est égal à $|S_0|$ **alors**
        assigner a_i à $\top$ dans le plan courant
        $\varphi \coloneqq \varphi'$
    sinon
        soit φ' une d-DNNF représentant $[\![\varphi]\!]_{|a_i=\bot}$ *// assigner la variable d'action à* $\bot$
        si le nombre de modèles de $\exists\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}.[\![\varphi']\!]$ est égal à $|S_0|$ **alors**
            assigner a_i à $\bot$ dans le plan courant
            $\varphi \coloneqq \varphi'$
        sinon
            revenir en arrière, en désinstanciant une des variables d'action

2.3.2 Planification par recherche heuristique

Bonet et Geffner [BG06] ont appliqué la compilation de connaissances à la planification d'une manière originale. Leur étude porte sur la planification déterministe comportant des pénalités et des récompenses associées aux fluents (l'approche classique consistant à les associer seulement aux actions) ; cela permet de modéliser des problèmes de planification sans but réel, mais seulement avec des préférences sur les résultats possibles, comme dans le cadre MDP.

L'heuristique présentée est relativement simple, et correspond schématiquement au coût du plan optimal pour le problème relaxé (c'est-à-dire en ignorant les

effets négatifs des actions). L'inconvénient de cette heuristique est qu'elle nécessite des calculs difficiles pour chaque état visité. Les auteurs ont donc proposé de compiler le problème en une d-DNNF booléenne, en utilisant divers encodages, dans le style de la planification par SAT ; les préférences sont simplement associées à chaque littéral de la formule résultante. En utilisant la forme compilée, les calculs de la valeur heuristique dans chaque état peuvent être effectués en temps linéaire (la preuve se base sur un résultat de Darwiche et Marquis [DM04], qui ont étudié la compilation de bases de connaissances pondérées, qui sort du cadre de cette thèse). Ils ont obtenu de bons résultats en pratique, bien que l'étape de compilation de certains de leurs *benchmarks* soit trop gourmande en temps ou en mémoire.

2.3.3 Planification par *model-checking*

Le paradigme de planification par *model-checking* est par nature très lié à la compilation de connaissances, et en particulier au langage des OBDDs.

Planification faible

Le premier algorithme de planification par *model-checking* symbolique, développé par Cimatti et al. [CG$^+$97], visait à résoudre des problèmes de planification faible. La procédure correspondante est décrite dans l'algorithme 2.4. Elle part des états initiaux, et construit récursivement l'ensemble des états atteignables. Elle s'arrête lorsque l'un des objectifs est atteint, ou quand elle détecte un point fixe, dans lequel plus aucun état n'est ajouté. En pratique, les auteurs ont utilisé le *model-checker* SMV [McM93], qui lui-même se base sur le langage $\texttt{OBDD}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$; cependant, notre algorithme 2.4 ne spécifie aucun langage. Il peut être appliqué à n'importe quel langage booléen, tant que les opérations correspondantes sont supportées, à savoir $\wedge$**BC**, **FO**, **CO** et **EQ**.

L'algorithme 2.4 se contente de vérifier si une solution existe ; pour en exhiber une, les auteurs lui ont adjoint une seconde procédure (non détaillée ici) qui remonte dans les ensembles d'états successifs.

Planification forte

Cimatti, Roveri et Traverso [CRT98b] ont utilisé l'approche de la planification par *model-checking* pour construire des plans forts, c'est-à-dire des plans garantis valides malgré le non-déterminisme. La procédure en question est différente de la précédente, car elle s'appuie sur une *recherche en arrière* : au lieu de calculer un ensemble d'états atteignables, elle construit un ensemble d'*états à atteindre*, en partant des buts. Tout d'abord, elle calcule l'ensemble des couples état-action garantissant qu'un des buts est atteint en une étape, ajoute les états correspondants à l'ensemble des « états à atteindre », puis calcule l'ensemble des couples état-action garantissant qu'un de ces états est atteint en une étape, etc. Nous présentons cette procédure dans l'algorithme 2.5 ; les auteurs l'ont implémentée avec des OBDDs booléens, mais une nouvelle fois, tout langage booléen supportant les opérations nécessaires peut faire l'affaire.

Algorithme 2.4 Algorithme de planification faible par *model-checking* avec recherche en avant.

entrée : un problème de planification $\langle\langle S, A, \gamma\rangle, S_0, S_g\rangle$
sortie : 1 s'il existe une solution faible, et 0 sinon
initialiser φ tel que $\text{Mod}(\varphi) = S_0$
initialiser G tel que $\text{Mod}(G) = S_g$
initialiser T tel que $\text{Mod}(T) = \{\, \langle \vec{s}, \vec{a}, \vec{s}' \rangle \in S \times A \times S' \mid \vec{s}' \in \gamma(\vec{s}, \vec{a}) \,\}$
répéter
    si $[\![\varphi]\!] \wedge [\![G]\!]$ est cohérent **alors** *// au moins un des états atteints est un état-but*
        renvoyer 1
    $\varphi_{\text{prec}} \coloneqq \varphi$
    soit φ' telle que $[\![\varphi']\!] \equiv \exists\mathcal{S}.\exists\mathcal{A}.([\![\varphi]\!] \wedge [\![T]\!])$ *// φ' est l'ensemble des états suivants atteignables*
    $\varphi \coloneqq \varphi'_{|\mathcal{S}\leftrightarrow\mathcal{S}'}$ *// les états suivants sont remplacés par les états courants*
jusqu'à $\varphi \equiv \varphi_{\text{prec}}$ *// on recommence tant que des états sont ajoutés*
renvoyer 0

Algorithme 2.5 Algorithme de planification forte par *model-checking* avec recherche en arrière.

entrée : un problème de planification $\langle\langle S, A, \gamma\rangle, S_0, S_g\rangle$
sortie : une politique-solution forte
initialiser I tel que $\text{Mod}(I) = S_0$
initialiser φ tel que $\text{Mod}(\varphi) = S_g$
initialiser T tel que $\text{Mod}(T) = \{\, \langle \vec{s}, \vec{a}, \vec{s}' \rangle \in S \times A \times S' \mid \gamma(\vec{s}, \vec{a}) = \vec{s}' \,\}$
initialiser δ à $\bot$
répéter
    si $\varphi \models I$ **alors** *// tous les états initiaux sont couverts*
        renvoyer δ
    $\varphi' \coloneqq \varphi_{|\mathcal{S}'\leftrightarrow\mathcal{S}}$ *// les états courants sont remplacés par les états suivants*
    $\delta_{\text{step}} \coloneqq \neg\varphi \wedge \forall\mathcal{S}'.(T \rightarrow \varphi')$ *// calculer les couples état-action garantissant d'arriver dans un état couvert*
    $\delta \coloneqq \delta \vee \delta_{\text{step}}$
    $\varphi_{\text{step}} \coloneqq \exists\mathcal{A}.\delta_{\text{step}}$ *// calculer l'ensemble des états nouvellement couverts*
    $\varphi \coloneqq \varphi \vee \varphi_{\text{step}}$
jusqu'à φ_{step} est incohérente *// on recommence tant que des états sont ajoutés*

L'algorithme 2.5 stocke dans δ la politique courante, et dans φ l'ensemble des « états couverts », c'est-à-dire des états depuis lesquels la politique est garantie de mener à l'un des buts. En effet, les états ajoutés à φ sont ceux pour lesquels il existe une action (ligne 13) telle que tous les états résultants sont déjà couverts (ligne 11).

2.3.4 Planification par MDPs

Hoey et al. [HS$^+$99] ont examiné la compilation de processus décisionnels de Markov. Leur approche est similaire à ce qui a été fait dans le cadre de la planification par *model-checking*, à savoir représenter un MDP en utilisant des variables booléennes pour exprimer l'état courant $\vec{s}$, l'action choisie $\vec{a}$ et l'état suivant $\vec{s}'$ obtenu après avoir appliqué l'action dans l'état courant. Cependant, dans le cas des MDPs, les transitions peuvent être valuées ; il n'est donc pas possible d'utiliser un langage booléen. Les auteurs ont employé le langage `ADD` [BF$^+$97] sur le domaine d'interprétation $\mathfrak{D}_{\mathcal{S}\cup\mathcal{A}\cup\mathcal{S}',[0,1]}$.

Ainsi, la valeur de l'ADD pour chaque instanciation est la probabilité associée au couple état-transition correspondant dans le MDP. Par exemple, en notant φ l'ADD et en considérant des instanciations $\vec{s}$, $\vec{a}$ et $\vec{s}'$, $[\![\varphi]\!](\vec{s}\ .\ \vec{a}\ .\ \vec{s}')$ est la probabilité d'arriver dans $\vec{s}'$ en appliquant $\vec{a}$ dans $\vec{s}$ — c'est-à-dire la valeur de $P_{\vec{a}}(\vec{s}, \vec{s}')$. En utilisant un autre ADD pour représenter la fonction de récompense, les auteurs présentent l'algorithme SPUDD (*stochastic planning using decision diagrams*, planification stochastique par diagrammes de décision) qui permet de résoudre un problème de planification par MDP en appliquant la procédure classique d'*itération de valeur* [Bel57]. L'intérêt de SPUDD est que les opérations sont faites sur des ADDs, et donc des ensembles d'états, plutôt que sur des états uniques, d'une façon similaire au paradigme de planification par *model-checking*. Cela a permis aux auteurs de résoudre des problèmes insolubles par les techniques classiques. Ce type d'approche a depuis été étendu à la recherche de politiques approchées [SHB00] et à la résolution de *MDPs du premier ordre* [JKK09].

*
**

Ce chapitre a survolé le domaine de la planification automatique, et a examiné plusieurs applications possibles de la compilation de connaissances au problème général qui en est l'objet. Dans le chapitre suivant, nous nous recentrons sur notre sujet, qui a notamment pour particularité d'impliquer des systèmes embarqués.

CHAPITRE

3

Problématique

À présent que nous avons présenté de façon générale la compilation de connaissances, la planification, et les relations entre ces deux domaines, nous nous concentrons sur notre sujet d'étude spécifique, à savoir l'application de la compilation de connaissances à des problèmes réalistes de contrôle de systèmes autonomes. Nous identifions plusieurs problèmes particuliers, pour lesquels la compilation pourrait s'avérer utile, et appliquons à l'un d'eux certaines techniques que nous avons présentées au chapitre précédent ; leurs inconvénients nous poussent alors à étudier de nouveaux langages-cibles, plus adaptés à la compilation de nos problèmes que les langages existants.

Nous décrivons tout d'abord les problèmes considérés [§ 3.1], puis présentons notre premier essai d'application de la compilation à ces problèmes [§ 3.2] ; enfin, nous détaillons l'orientation générale de la thèse [§ 3.3].

3.1 *Benchmarks* utilisés

Nous considérons quatre *benchmarks* que nous estimons représentatifs de problèmes réalistes de prise de décision pour des systèmes embarqués. Cette section les décrit informellement ; leur spécification complète peut être trouvée soit dans l'article d'où ils ont été tirés, soit dans les annexes de la version longue de la thèse.

3.1.1 Problème de compétition de drones

Le problème *Drone* traite de la gestion des objectifs d'un micro-drone dans une compétition[1]. Ce genre de compétition implique généralement des drones devant

[1] Par exemple, la compétition organisée au sein de la *International Micro Air Vehicle Conference* ; voir http://www.imav2011.org/.

accomplir un certain nombre de buts. Dans notre problème *Drone*, adapté de Verfaillie et Pralet [VP08b] (voir la version longue de la thèse pour les spécifications de notre version), le terrain est divisé en *zones* contenant chacune une cible. Il y a trois sortes de cibles différentes, chaque sorte rendant compte d'une capacité spécifique du drone.

- Cibles à *identifier* : le drone doit pouvoir par exemple déterminer la couleur de la cible. Cela implique d'effectuer une manœuvre spéciale (un « huit ») au dessus de la cible.
- Cibles à *localiser* : le drone n'a aucune information sur la localisation précise de la cible dans sa zone, et doit la trouver. Cela implique une manœuvre de « balayage » de la zone toute entière.
- Cibles à *toucher* : le drone lâche une bille qui doit atteindre la cible, dont les coordonnées sont connues.

Il y a également une zone spéciale, la « base », de laquelle le drone décolle et dans laquelle il doit avoir atterri à la fin de la mission. L'objectif général du drone est de décoller, d'accomplir tous les objectifs et d'atterrir, le tout dans le temps alloué.

Le problème est formulé comme un problème de planification classique avec des fluents, sous la forme de réseaux de contraintes (un pour les préconditions des actions et un pour leurs effets, plus des contraintes représentant les états initial et but). Ces réseaux de contraintes portent sur des variables discrètes, mais aussi une variable continue qui exprime la variable d'état « temps restant ».

On voudrait pouvoir compiler ces réseaux de contraintes pour pouvoir leur appliquer une approche de type planification par satisfiabilité [§ 2.3.1.1] ou planification par *model-checking* [§ 2.3.3].

3.1.2 Problème de gestion de la mémoire d'un satellite

Le second problème, *ObsToMem* [PV$^+$10], concerne l'organisation des connections entre l'instrument d'observation et la mémoire de masse d'un satellite. L'instrument d'observation consiste en un ensemble de capteurs organisés en *lignes de détection*. L'information enregistrée par chaque ligne doit être compressée ; cela est effectué par des *compresseurs de mémoire (COMs)*. Chaque ligne de détecteurs ne peut être connectée qu'à certains COMs seulement — il y a généralement un nombre différent de lignes et de COMs. Si, à un moment donné, aucun COM n'est disponible pour compresser la sortie d'une ligne de détection, des données sont perdues ; pour éviter cela, le satellite contient plus de COMs que de lignes, chaque COM étant associé à un ensemble fixé de lignes.

L'information compressée est alors stockée dans la mémoire de masse du satellite. Cette mémoire est divisée en *banques* ; chaque COM ne peut écrire que dans certaines banques mémoire. En résumé, pour que les informations soient correctement enregistrées, il faut que chaque ligne de détection soit connectée à un

COM disponible, qui doit lui-même être connecté à une banque mémoire disponible. L'objectif de ce *benchmark* est de construire un *contrôleur* garantissant (autant qu'il est possible) que les données soient inscrites dans la mémoire de masse, même en cas de dysfonctionnement de COMs ou de banques mémoire. Dans ce but, le contrôleur doit pouvoir modifier les configurations des connections entre tous ces éléments.

ObsToMem est donc un *problème de contrôle* : sa solution est une politique de décision garantissant que les exigences sont remplies en permanence. Il se présente sous la forme de plusieurs réseaux de contraintes, qui représentent la relation de transition, la propriété de sûreté (c'est-à-dire l'objectif permanent), et l'état initial. Ces réseaux de contraintes n'impliquent aucune variable continue, mais un grand nombre de variables énumérées, dont la taille du domaine dépend du nombre de lignes de détection, de COMs, et de banques mémoire.

Nous voudrions compiler ces réseaux de contraintes, de manière à pouvoir appliquer par exemple la planification par *model-checking* [§ 2.3.3].

3.1.3 Problème de gestion des connections d'un transpondeur

Le problème *Telecom* (voir la version longue de la thèse pour la spécification formelle) vise également à construire un contrôleur gérant les connections entre divers éléments d'un satellite. L'objet de ce *benchmark* est le *transpondeur* d'un satellite de communications, c'est-à-dire la série d'éléments formant le canal de communication entre les antennes de réception et d'émission du satellite. Dans ce problème, on ne considère que trois éléments : les canaux d'*entrée* et de *sortie*, et les amplificateurs de signal. L'objectif est de construire un contrôleur garantissant que les données d'entrée sont correctement amplifiées et connectées à un canal de sortie, même si certains canaux ou amplificateurs sont en panne. Le problème est modélisé différemment du problème *ObsToMem*, en ce que les configurations possibles sont données sous la forme d'un ensemble de *chemins* reliant canaux d'entrée, amplificateurs, et canaux de sortie.

Le problème est spécifié par un réseau de contraintes sur des variables d'état et d'action, qui représentent respectivement quels appareils fonctionnent ou non, et quel chemin est assigné à chaque canal d'entrée. Les solutions du réseau de contraintes sont des configurations connectant des canaux d'entrée à des canaux de sortie fonctionnels en passant par des amplificateurs fonctionnels.

Nous voudrions compiler ce réseau de contraintes, soit pour l'utiliser directement en ligne (chaque fois qu'une panne est détectée, une nouvelle configuration est recherchée), soit pour construire une politique.

3.1.4 Problème de rendez-vous en attitude

Le *benchmark Satellite* fait partie d'un problème de prise de décision pour un satellite agile d'observation de la Terre, équipé d'un instrument de détection de nuages [BVC07]. Le satellite doit prendre des décisions très rapides en fonction

de la couverture nuageuse des zones qu'il est censé observer. Une des décisions possibles est « pointer vers le Soleil pour recharger les batteries ». Cette manœuvre prenant un certain temps, elle ne doit pas être utilisée à tort et à travers ; l'observation suivante doit notamment être suffisamment lointaine dans le temps.

Calculer le temps nécessaire à la complétion de la manœuvre de « pointage Soleil » n'est pas une tâche triviale. Elle se base sur des équations portant sur un certain nombre de paramètres continus, comme les angles représentant l'attitude courante du satellite. Le problème *Satellite* consiste à décider s'il est possible d'effectuer cette manœuvre dans le temps imparti. Le résultat est alors utilisé comme un paramètre par le programme principal de prise de décision embarqué dans le satellite.

Notre objectif est de compiler le réseau de contraintes représentant ce sous-problème, de façon à ce que le programme de prise de décision s'appuie sur la forme compilée et obtienne la réponse aussi rapidement que possible. On peut voir cela comme une *compilation partielle*, en ce qu'on ne compile qu'une partie du problème, et utilise des techniques différentes pour les autres parties.

3.2 Première tentative

3.2.1 Notre approche pour le problème *Drone*

Nous avons commencé notre étude en essayant d'appliquer un algorithme de planification forte par *model-checking* [§ 2.3.3.2] au problème *Drone*. D'un point de vue théorique, les problèmes de planification forte et faible sont identiques dans ce cas-là, puisque les actions sont déterministes ; cependant, construire une politique au lieu d'un plan est plus robuste, étant donné qu'elle peut résoudre certaines imperfections du modèle déterministe en pratique, au sens où tant que l'état courant est couvert par la politique, le système autonome peut atteindre le but. De plus, en utilisant l'algorithme de planification forte qui part du but et ajoute itérativement des états couverts, il est possible de construire une politique couvrant même des états théoriquement impossibles à rencontrer.

Nous avons implanté cet algorithme, en le modifiant de façon à ce qu'au lieu de s'arrêter quand tous les états initiaux sont couverts, il continue jusqu'à atteindre un point fixe, c'est-à-dire un point où il est garanti que tous les états desquels le but est atteignable sont couverts par la politique. Les états restants sont considérés comme des « culs-de-sacs ».

Notre procédure compile chaque réseau de contraintes (à savoir les préconditions et les effets des actions, et l'objectif) vers $\texttt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}}$. Elle construit alors la politique-solution sous la forme d'un OBDD, de la manière décrite précédemment [voir aussi GT99]. La figure 3.1 montre la politique résultante pour l'instance numéro 1. Pour pouvoir compiler des CNs portant sur la variable réelle (le « temps restant »), la procédure discrétise arbitrairement le domaine de cette variable en

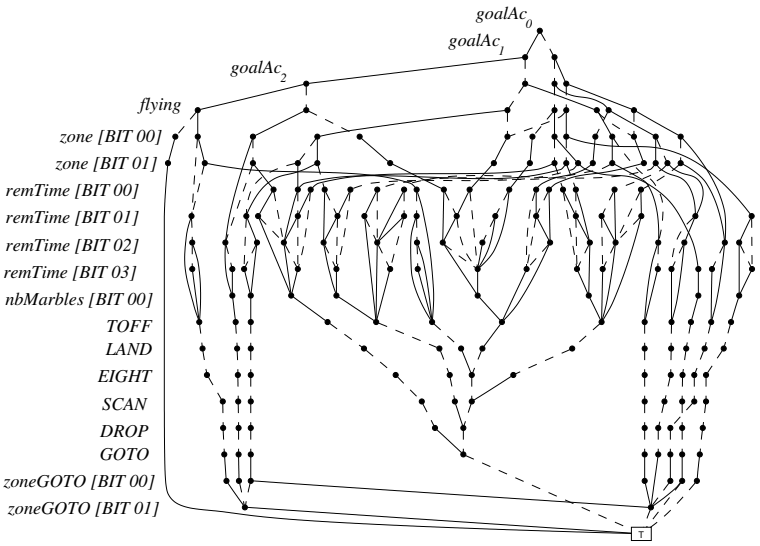


Fig. 3.1 : Exemple de politique pour le problème *Drone*, sous la forme d'un OBDD, obtenu grâce à la procédure de planification par *model-checking*. Il s'agit de l'instance numéro 1, avec 4 zones. Le nœud $\bot$ et tous les arcs y pointant ne sont pas représentés.

petites unités de temps, la transformant ainsi en une variable énumérée de domaine suffisamment large pour que les résultats restent significatifs malgré la perte de précision. Cette variable énumérée est ensuite, à l'instar des autres variables à domaine entier, remplacée par un certain nombre de variables booléennes, chacune correspondant à un bit donné de la représentation binaire des valeurs du domaine. Cette transformation est connue sous le nom de *« log encoding »* [voir e.g. SK$^+$90, Wal00].

3.2.2 Résultats pour le problème *Drone*

Le tableau 3.1 montre des résultats obtenus en utilisant cette approche sur une version simplifiée du problème *Drone* (toutes les actions ont la même durée arbitraire). Nous avons considéré différentes instances de ce problème, en faisant varier le nombre d'objectifs : l'instance n comprend ainsi n zones par type d'objectif, soit $3n + 1$ zones au total (base incluse).

En ce qui concerne la politique obtenue, l'approche s'avère fructueuse : une politique comportant une centaine de milliers de nœuds peut vraisemblablement être embarquée. Cependant, le plus grand OBDD utilisé durant la construction de la politique contenait plus de 3.5 millions de nœuds ; l'existence de ces structures intermédiaires nous a empêché de compiler l'instance $n = 6$, en raison d'un espace mémoire insuffisant.

instance	politique	entrées	plus grand
0	29	4274	4225
1	383	63 869	85 411
2	1219	164 176	323 222
3	4547	248 631	661 505
4	16 494	361 043	1 241 057
5	144 087	781 995	3 595 802

Tab. 3.1 : Nombre de nœuds de divers OBDDs — politique-solution, réseaux de contraintes d'entrée, et plus grand OBDD rencontré au cours de la recherche — pour six instances du problème *Drone*.

3.3 Vers des langages-cibles plus appropriés

Dans cette section, nous décrivons brièvement la problématique que nous suivons dans ce travail, en nous appuyant sur les caractéristiques de notre sujet d'étude.

3.3.1 Orientation générale

Au lieu de compiler nos problèmes vers `OBDD`, nous décidons d'étudier la possibilité d'utiliser un langage-cible plus approprié. En particulier, nous voudrions qu'il puisse représenter des réseaux de contraintes portant à la fois sur des variables booléennes, énumérées et continues.

Il existe plusieurs langages booléens sur des variables *énumérées*, parmi lesquels `MDD` et d'autres diagrammes de décision. Cependant, il n'existe aucune carte de compilation de ces langages, et qui plus est, nous n'avons trouvé aucune étude portant sur la compilation de problèmes impliquant des variables *continues*.

Nous décidons donc de définir un langage sur des variables continues, en nous inspirant de la famille des diagrammes de décision. Néanmoins, nous ne voulons pas directement transposer les diagrammes de décision binaires ordonnés ou les diagrammes de décision multivalués dans un contexte continu ; en effet, ces langages supportent plus de requêtes et de transformations que nécessaire pour nos applications. Nous commençons par définir un langage général, puis le restreignons de manière à ce qu'il satisfasse l'ensemble des requêtes et transformations dont nous avons besoin, tout en restant aussi compact que possible.

3.3.2 Identification des opérations importantes

Pour pouvoir chercher des langages aussi généraux que possible mais adaptés à nos problèmes, nous listons les requêtes et transformations qu'un langage doit satisfaire pour que nos applications soient polynomiales. En fonction du problème, nous avons deux objectifs différents :

- soit compiler une solution du problème, sous la forme d'une politique ;
- soit compiler une partie du problème (typiquement une relation de transition) pour rendre la résolution en ligne polynomiale.

Manipulation de politiques de décision

Dans les problèmes *Drone* et *ObsToMem*, nous voulons compiler une politique de décision, c'est-à-dire une fonction associant à tout état un ensemble d'actions adaptées — ou, de manière équivalente, une fonction associant tout couple $\langle s, a \rangle$ à une valeur booléenne indiquant si faire l'action a dans l'état s est un bon choix. Une telle fonction peut être compilée dans un langage de représentation booléen.

Pour exploiter la politique en ligne, deux opérations de base sont nécessaires. En premier lieu, à chaque fois qu'un nouvel état est observé — correspondant à une certaine instanciation des variables d'état — l'ensemble des actions adaptées doit être calculé. Cela correspond à un *conditionnement* de la forme compilée par l'instanciation en question. La structure obtenue représente l'ensemble des « bonnes » décisions ; selon les cas, il peut être utile de toutes les lister, ce qui correspond à une *énumération de modèles*, ou de simplement en choisir une, ce qui constitue une *extraction de modèle*. Contrairement aux deux premières, cette dernière opération n'est pas définie dans la carte de compilation existante. Nous l'introduisons dans la section suivante [§ 3.3.3].

Manipulation de relations de transition

Dans tous nos *benchmarks*, excepté *Satellite*, nous devons manipuler des relations de transition, c'est-à-dire des relations liant un état donné s et une action donnée a à l'état s' résultant de l'application de a dans s. Ces relations sont représentables comme des fonctions booléennes sur des variables d'état et d'action. Une fois compilées, elles peuvent être manipulées de diverses manières.

La plus simple consiste à calculer, pour un état courant s et une action a, l'ensemble de tous les états successeurs possibles. Pour cela, il est nécessaire de conditionner la forme compilée par l'instanciation représentant s et a, puis d'énumérer les modèles de la structure obtenue. Un autre usage est de calculer l'ensemble de tous les états successeurs possibles, indépendamment de l'action effectuée ; dans ce cas, il est de nouveau nécessaire de conditionner par s, mais l'étape suivante est d'*oublier* les variables d'action, avant d'énumérer les modèles.

Une utilisation intéressante d'une relation de transition compilée est la construction d'une politique de décision, par l'application d'une approche « planification par *model-checking* » (par exemple la procédure décrite dans la section 3.2). Les principales opérations nécessaires sont la conjonction et la disjonction bornées, la négation, l'oubli, et l'enforcement (cette dernière n'étant pas nécessaire si toutes les actions sont déterministes).

3.3.3 Nouvelles requêtes et transformations

La manipulation en ligne de politiques de décision compilées nécessite une opération d'*extraction de modèle*. Comme elle n'est pas mentionnée dans la littérature sur la carte de compilation, nous l'introduisons ici, en même temps que quelques autres intéressantes nouvelles requêtes et transformations.

Définition 3.3.1 (Requêtes d'extraction)**.** Soit L un sous-langage de GRDAG.

- L satisfait **MX** (*model extraction*, extraction de modèle) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ à un de ses modèles s'il en existe un, et s'arrête sans rien retourner sinon.
- L satisfait **CX** (*context extraction*, extraction de contexte) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et toute variable y à $\text{Ctxt}_\varphi(y)$.

L'extraction du contexte d'une variable, c'est-à-dire l'ensemble de ses valeurs cohérentes, est particulièrement intéressant quand son domaine n'est pas booléen. Cette opération peut être par exemple utilisée pour étudier une propriété des états successeurs dans une relation de transition ; elle a également des applications en configuration (quelles valeurs sont toujours disponibles pour ce paramètre ?) et en diagnostic (quelles sont les pannes possibles de ce composant ?).

Intéressons-nous à présent au conditionnement ; comme expliqué dans la section 1.4.1.3, il peut être généralisé de deux façons aux variables non-booléennes. Nous avons choisi de garder l'idée que conditionner une fonction signifie assigner des valeurs à certaines variables. Mais il est parfois nécessaire de restreindre ces variables à un sous-ensemble de leur domaine, pas forcément à une unique valeur ; c'est le rôle de la transformation suivante.

Définition 3.3.2 (Restriction à un terme)**.** Soit L un sous-langage de GRDAG. L satisfait **TR** (*term restriction*, restriction à un terme) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et tout terme cohérent γ de L à une L-représentation de la restriction $[\![\varphi]\!]_{|[\![\gamma]\!]}$ de $[\![\varphi]\!]$ à $[\![\gamma]\!]$.

La restriction à un terme est donc une extension du conditionnement. Quand elles sont appliquées à des variables booléennes, ces deux transformations sont complètement équivalentes, comme nous l'avons expliqué en section 1.4.1.3. La restriction à un terme peut être utilisée, par exemple, sur une forme compilée représentant une politique de décision : si l'observation de l'état courant n'est pas précise, au lieu de conditionner les variables d'état, elles peuvent être restreintes. La structure résultante représente alors l'ensemble des actions qui pourraient convenir à l'état courant non connu avec précision. Bien sûr, **TR** est fortement liée à **CD**, et également à **FO**, par définition.

La restriction à un terme est également liée à la conjonction, mais pas celle de fonctions booléennes quelconques. Il s'agit d'une conjonction avec un terme, suivie de l'oubli des variables de ce terme. En général, il est facile de conjoindre

une structure avec un terme, même si le langage en question ne satisfait pas $\wedge$**C**, ni même $\wedge$**BC**. Comme cette propriété est souvent utilisée dans les preuves, nous introduisons la transformation spécifique qui correspond, celle de « conjonction avec un terme », et pour des raisons similaires, sa duale « disjonction avec une clause ».

Définition 3.3.3. Soit L un sous-langage de GRDAG.

- L satisfait $\wedge$**tC** (*closure under conjunction with a term*, fermeture par la conjonction avec un terme) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et tout terme γ de L à une L-représentation de $[\![\varphi]\!] \wedge [\![\gamma]\!]$.
- L satisfait $\vee$**clC** (*closure under disjunction with a clause*, fermeture par la disjonction avec une clause) si et seulement s'il existe un algorithme polynomial associant toute L-représentation φ et toute clause γ de L à une L-représentation de $[\![\varphi]\!] \vee [\![\gamma]\!]$.

*
**

En résumé, nous décidons d'étudier la possibilité de compiler nos problèmes en utilisant des langages-cibles plus adaptés, capables de manipuler conjointement variables discrètes et variables continues. Nous restons dans le cadre de GRDAG, mais nous efforçons de rester aussi général que possible, dans le but de maintenir une compacité maximale. Les requêtes et transformations [voir définitions 1.4.12 et 1.4.13] que les nouveaux langages doivent satisfaire sont les suivantes :

- **CD** et **MX** — obligatoires, pour les manipulations de base des politiques de décision et des tables de transition ;
- **ME** et **FO** — souhaitables, pour pouvoir manipuler ces éléments plus finement ;
- $\wedge$**BC**, $\vee$**BC**, $\neg$**C** — nécessaires à l'application de notre algorithme basé sur la planification forte par *model-checking* ;
- **EN** — souhaitable, pour que ledit algorithme puisse manipuler des actions non-déterministes.

Deuxième partie

Automates à intervalles

Introduction

En essayant d'appliquer la compilation de connaissances à un des problèmes de planification que nous cherchons à traiter, nous avons été confrontés à l'écueil des *variables réelles*, qui ne sont habituellement pas considérées dans le cadre de la compilation — bien qu'elles soient très utiles dans la représentation de problèmes réalistes. Les paramètres continus, tels que le temps ou l'énergie, sont en général arbitrairement *discrétisés* pour pouvoir être exprimés par des variables énumérées. Ainsi, nous avons représenté le paramètre « temps restant » du problème *Drone* par le biais d'une variable entière, réduisant de fait la précision à un nombre constant de secondes, fixé au préalable. Cette méthode conduit à manipuler des variables à grand domaine, ce qui a un impact sur la taille des structures compilées.

Nous avons décidé d'étudier l'efficacité de langages-cibles de compilation permettant d'utiliser des variables continues — ainsi que des variables à grand domaine énuméré — sans nécessiter de discrétisation arbitraire. Les diagrammes à intervalles (*interval diagrams*) [ST98] semblaient un bon point de départ, mais ils n'ont en pratique jamais été appliqués à des variables continues, ni utilisés pour la planification. De plus, étant destinés à une utilisation dans le domaine du *model-checking* symbolique, ils sont suffisamment contraints pour supporter la requête d'équivalence, dont nous n'avons pas besoin. Nous avons donc tenté de relâcher certaines des restrictions structurelles des diagrammes à intervalles, en définissant un langage plus général, que nous avons nommé *automates à intervalles*.

CHAPITRE

4

Formalisme des automates à intervalles

Dans ce chapitre, nous les *automates à intervalles*, un langage-cible de compilation pour les problèmes impliquant des variables continues. Nous identifions par la suite une restriction structurelle des automates à intervalles, leur permettant de supporter des requêtes et transformations fréquemment utilisées, en particulier celles nécessaires à la planification.

Ce chapitre est divisé en trois sections : nous commençons par définir le langage général des automates à intervalles [§ 4.1], puis introduisons le sous-langage des automates à intervalles convergents [§ 4.2], et enfin présentons la carte de compilation de la famille des automates à intervalles [§ 4.3].

4.1 Langage

4.1.1 Définition

Le langage des automates à intervalles peut être défini comme un sous-langage de `GRDAG` ; introduisons quelques notions au préalable.

Définition 4.1.1 (Intervalle)**.** Soit S un ensemble totalement ordonné par $\leqslant$. $A \subseteq S$ est un *intervalle de* S si et seulement s'il s'agit d'une partie convexe de S, c'est-à-dire que $\forall \langle x, y \rangle \in A^2, \forall z \in S, x \leqslant z \leqslant y \implies z \in A$.

Un intervalle est dit *borné à gauche* (resp. *borné à droite*) si et seulement s'il a un minorant (resp. un majorant), c'est-à-dire que $\exists m \in S, \forall x \in A, m \leqslant x$ (resp. $x \leqslant m$).

On appelle simplement *borne gauche* (resp. *borne droite*) le plus grand minorant (resp. majorant) d'un intervalle borné à gauche (resp. à droite).

Un intervalle est dit *ouvert à gauche* (resp. *ouvert à droite*) si et seulement s'il

est borné à gauche (resp. à droite) et qu'il ne contient pas sa borne gauche (resp. droite), c'est-à-dire que $\forall x \in A, \exists m \in A, m < x$ (resp. $x < m$).

Un intervalle *fermé* est un intervalle qui n'est ouvert ni à gauche, ni à droite.

Notons qu'un intervalle fermé n'est pas nécessairement fini : $\mathbb{R}_+$ est un intervalle fermé de $\mathbb{R}$. On s'intéressera principalement aux intervalles de $\mathbb{R}$, et on notera $[a, b]$ l'intervalle fermé de bornes a et b, $[a, +\infty]$ l'intervalle fermé de borne gauche a et n'ayant pas de borne droite, et $[-\infty, b]$ l'intervalle fermé de borne droite b et n'ayant pas de borne gauche. On note $\mathbb{I}S$ l'ensemble des intervalles fermés d'un ensemble S. On utilisera parfois des intervalles non fermés ; précisons dès à présent qu'on les note en utilisant des crochets inversés, comme $[a, b[$ pour l'intervalle correspondant à $[a, b] \setminus \{b\}$.

Définition 4.1.2 (Pavabilité)**.** Soit S un ensemble totalement ordonné, et $A \subseteq S$. A est *S-pavable* si et seulement s'il existe $n \in \mathbb{N}$ et une suite $\langle A_1, \ldots, A_n \rangle \in (\mathbb{I}S)^n$ tels que $A = \bigcup_{i=1}^{n} A_i$.

Un ensemble S-pavable est donc simplement une union finie d'intervalles fermés de S. On note $\mathbb{T}S$ l'ensemble de tous les ensembles S-pavables (*tileable*). Ainsi, $\mathbb{T}\mathbb{R}$ contient bien sûr tous les intervalles fermés et toutes les parties finies de $\mathbb{R}$; mais il ne contient pas $\mathbb{N}$, par exemple. Cette notion va nous être utile pour définir la catégorie de variables à laquelle on s'intéresse dans cette deuxième partie ; on l'utilisera à nouveau plus tard [§ 7.1.1].

Dans cette partie, on cherche à manipuler des variables réelles, ainsi que des variables énumérées. Nous choisissons d'utiliser des variables $\mathbb{R}$-pavables pour couvrir les deux possibilités. Notons $\mathcal{T} = \mathcal{V}_{\mathbb{T}\mathbb{R}}$ l'ensemble des variables $\mathbb{R}$-pavables ; cet ensemble a l'intéressante propriété d'inclure à la fois les variables à domaine énuméré fini, comme $\{1, 3, 56, 4.87\}$, et celles à domaine continu, comme $[1, 7] \cup [23.4, 28]$.

Les domaines $\mathbb{R}$-pavables sont également intéressants en raison de leur capacité à être représentés par des structures de données très simples (contrairement à une union *infinie* d'intervalles, par exemple), typiquement une liste de bornes. Définissons à présent la *taille caractéristique* d'un ensemble pavable, qui doit être représentative de la taille prise par la structure de données en mémoire, indépendamment de l'implémentation [§ 1.2.2.2].

Définition 4.1.3 (Taille d'un ensemble pavable)**.** Soit $S \in \mathbb{T}\mathbb{R}$ un ensemble $\mathbb{R}$-pavable. La taille caractéristique de S est définie comme le plus petit nombre d'intervalles nécessaires pour recouvrir S :

$$\|S\| = \min \left\{ n \in \mathbb{N} \;\middle|\; \exists \langle A_1, \ldots, A_n \rangle \in (\mathbb{I}\mathbb{R})^n, S = \bigcup_{i=1}^{n} A_i \right\}.$$

Nous pouvons maintenant donner la définition formelle de notre langage.

Définition 4.1.4. On définit le langage `IA` comme la restriction de $\mathtt{NNF}_{\mathcal{T}}^{\mathbb{I}\mathbb{R}}$ aux représentations satisfaisant la décision $\wedge$-simple [déf. 1.3.23].

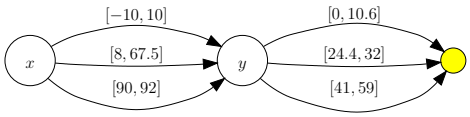


Fig. 4.1 : Exemple d'automate à intervalles. Son ensemble de modèles [déf. 1.4.5] est $[-10, 10] \times [0, 10.6] \cup [-10, 10] \times [24.4, 32] \cup [-10, 10] \times [41, 59] \cup [8, 67.5] \times [0, 10.6] \cup [8, 67.5] \times [24.4, 32] \cup [8, 67.5] \times [41, 59] \cup [90, 92] \times [0, 10.6] \cup [90, 92] \times [24.4, 32] \cup [90, 92] \times [41, 59]$.

En d'autres termes, les IA-représentations sont des GRDAGs sur des variables $\mathbb{R}$-pavables, avec des littéraux de la forme « $x \in [a, b]$ », et les restrictions structurelles de la famille des « diagrammes de décision » [§ 1.3.5]. Le nom IA signifie *interval automata* (automate à intervalle) ; cependant, suivant en cela les usages pour les BDDs et les automates à états finis, nous ne manipulerons pas les automates à intervalle sous cette forme de GRDAG. Dans la section suivante, nous introduisons la représentation « diagramme de décision » des automates à intervalles, équivalente à la forme GRDAG, mais permettant d'en simplifier la manipulation.

4.1.2 Automates à intervalles

La définition suivante décrit les automates à intervalles dans leur forme usuelle.

Définition 4.1.5. Un *automate à intervalles* (IA) est un multigraphe acyclique orienté avec au plus une racine et au plus une feuille (le *puits*), dont les nœuds internes sont étiquetés par une variable de $\mathcal{T}$ ou par le symbole disjonctif $\veebar$, et dont les arcs sont étiquetés par un intervalle fermé de $\mathbb{R}$.

Avec cette définition, les automates à intervalles sont différents des éléments du langage IA introduit dans la section précédente : IA est défini comme un sous-langage de NNF, alors que la structure de la définition 4.1.5 n'est clairement pas une NNF. Cela est cependant sans conséquence, étant donné qu'il existe une bijection entre ces deux structures, exactement comme pour les BDDs [§ 1.3.5].

Dans la suite, nous considérerons toujours implicitement que les automates à intervalles sont sous cette forme. La figure 4.1 donne un exemple d'automate à intervalle et de son ensemble de modèles. Ce dernier a une forme particulière, celle d'un produit cartésien d'intervalles fermés, que nous appelons « union de boîtes ». Cela n'est pas surprenant, eu égard à la structure des automates à intervalles : ils se comportent exactement comme les BDDs, leur ensemble de modèles étant ainsi composé de toutes les instanciations compatibles avec au moins un chemin. Comme chaque chemin représente un *terme*, par exemple $\big[x \in [-10, 10]\big] \wedge \big[y \in [0, 10.6]\big]$ pour le chemin du haut de la figure 4.1, l'ensemble des instanciations compatibles ne peut être qu'une boîte. Par conséquent, un IA contenant toujours un nombre fini de chemins, son ensemble de modèles ne peut être qu'une union finie de boîtes.

Grâce à cette remarque, il est d'ores et déjà possible de montrer qu'IA n'est pas un langage complet [définition 1.2.11], puisqu'ilsne peut pas représenter de fonction dont l'ensemble de modèles est une union infinie de boîtes, comme par exemple $\big[x \in \mathbb{N}\big]$.

Proposition 4.1.6. Le langage IA n'est pas complet.

Notons que la fonction booléenne renvoyant toujours $\top$ peut être représentée par l'automate-puits, de la même manière que pour les BDDs. Cependant, comme les IAs n'ont qu'une seule feuille, la fonction booléenne renvoyant toujours $\bot$ peut être représentée par l'automate *vide*. Cela est permis par la définition, et est cohérent avec la fonction d'interprétation des diagrammes de décision [§ 1.1.2.1] : comme « la fonction renvoie vrai si et seulement s'il existe un chemin compatible », elle ne renvoie jamais vrai, puisqu'il n'y a aucun chemin.

On traite le symbole disjonctif $\veebar$ comme une variable spéciale, en imposant arbitrairement $\mathrm{Dom}(\veebar) = \{0\}$. Soit φ un IA, N un nœud et E un arc de φ. On peut alors définir les éléments suivants :

- $\mathrm{Root}(\varphi)$ la racine de φ et $\mathrm{Sink}(\varphi)$ son puits ;
- $\mathrm{Var}(N)$ la variable étiquetant N (par convention $\mathrm{Var}(\mathrm{Sink}(\varphi)) = \veebar$) ;
- $\mathrm{Lbl}(E)$ l'intervalle étiquetant E ;
- $\mathrm{Var}(E) = \mathrm{Var}(\mathrm{Src}(E))$ la variable associée à E.

Comme suggéré par la figure 4.1, la taille d'un automate peut être exponentiellement inférieure à la taille de son ensemble de modèles (exprimé comme une union de boîtes). Cela vient notamment du fait que les IAs peuvent être *réduits* par suppression des redondances, à la manière des BDDs et des NNFs. Avant de détailler l'opération de réduction, précisions les relations qui existent entre les IAs et les autres structures susnommées.

4.1.3 Relations avec la famille de BDD

Les automates à intervalles peuvent être vus comme une généralisation des diagrammes de décision binaires. L'interprétation des BDDs est en effet similaire à celle des IAs : pour une instanciation donnée des variables, la valeur de la fonction est $\top$ si et seulement s'il existe une chemin de la racine à la feuille $\top$ tel que l'instanciation en question est cohérente avec chacun des arcs le long du chemin. Si on les restreint à une même catégorie de variables, les BDDs sont même des IAs particuliers.

Proposition 4.1.7. $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{IA}$.

En particulier, comme on considère que $\mathbb{B}$ est un sous-ensemble de $\mathbb{N}$, les variables booléennes sont pavables (c'est-à-dire que $\mathcal{B} \subseteq \mathcal{T}$) et bien sûr $\mathbb{SB} \subseteq \mathbb{IR}$; on obtient alors le résultat suivant.

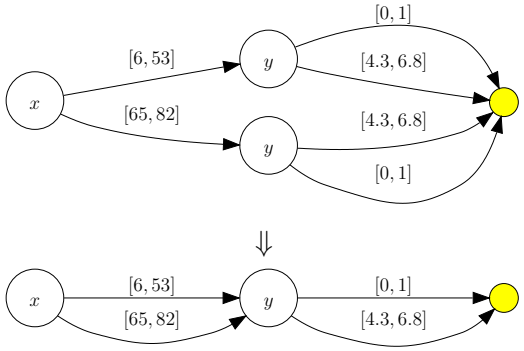


Fig. 4.2 : Fusion de nœuds isomorphes.

Corollaire 4.1.8. $\mathtt{BDD}_{\mathcal{B}}^{\mathbb{SB}} \subseteq \mathtt{IA}$.

Ainsi, les automates à intervalles sont plus généraux que les diagrammes de décision binaire, et ce pour trois raisons :

- ils autorisent des variables et des étiquettes plus générales ;
- ils autorisent les nœuds de décision non-exclusifs, et peuvent donc être non-déterministes ;
- ils autorisent les nœuds purement disjonctifs.

4.1.4 Réduction

Comme pour un BDD, on peut réduire la taille d'un automate à intervalles sans changer sa sémantique en fusionnant des nœuds et des arcs. Les opérations de réduction que l'on introduit ici sont basées sur les notions de nœuds *isomorphes*, *bégayants* et *non décisifs*, et d'arcs *contigus* et *morts*. Certaines de ces notions sont des généralisations directes des définitions introduites dans le contexte des BDDs (booléens) [Bry86], alors que les autres sont spécifiques aux automates à intervalles.

Définition 4.1.9 (Nœuds isomorphes)**.** Deux nœuds internes N_1 et N_2 d'un IA φ sont *isomorphes* si et seulement si

- $\mathrm{Var}(N_1) = \mathrm{Var}(N_2)$;
- il existe une bijection σ de $\mathrm{Out}(N_1)$ vers $\mathrm{Out}(N_2)$, telle que pour chaque arc $E \in \mathrm{Out}(N_1)$, $\mathrm{Lbl}(E) = \mathrm{Lbl}(\sigma(E))$ et $\mathrm{Dest}(E) = \mathrm{Dest}(\sigma(E))$.

Les nœuds isomorphes sont redondants, en ce qu'ils représentent la même sous-fonction ; seul l'un des deux est vraiment nécessaire (voir la figure 4.2). Cela correspond à la procédure habituelle sur les BDDs (et plus généralement sur les langages de type DAG).

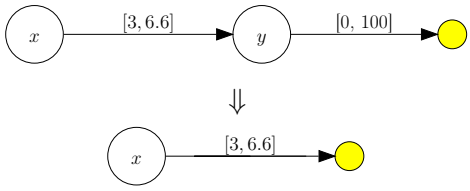


Fig. 4.3 : Élimination d'un nœud non décisif.

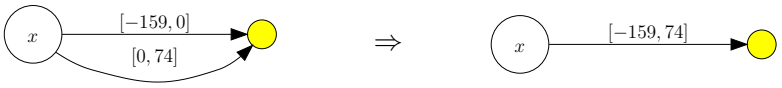


Fig. 4.4 : Fusion d'arcs contigus.

Définition 4.1.10 (Nœuds non décisifs)**.** Un nœud N d'un IA φ est *non décisif* si et seulement si $|\text{Out}(N)| = 1$ et $E \in \text{Out}(N)$ vérifie $\text{Dom}(\text{Var}(E)) \subseteq \text{Lbl}(E)$.

Un nœud non décisif ne restreint pas les solutions correspondant aux chemins qui le traversent ; il est toujours « automatiquement » franchi (voir figure 4.3).

Définition 4.1.11 (Arcs contigus)**.** Deux arcs E_1 et E_2 d'un IA φ sont *contigus* si et seulement si :

- $\text{Src}(E_1) = \text{Src}(E_2)$;
- $\text{Dest}(E_1) = \text{Dest}(E_2)$;
- il existe un intervalle $A \subseteq \mathbb{R}$ tel que $A \cap \text{Dom}(\text{Var}(E_1)) = (\text{Lbl}(E_1) \cup \text{Lbl}(E_2)) \cap \text{Dom}(\text{Var}(E_1))$.

Deux arcs contigus sortent d'un même nœud, pointent vers un même nœud, et ne sont pas disjoints (modulo le domaine de leur variable) : ils peuvent être remplacés par un seul arc (voir figure 4.4). Par exemple, dans le cas d'une variable entière, un couple d'arcs respectivement étiquetés $[0, 3]$ et $[4, 8]$ est équivalent à un seul arc étiqueté $[0, 8]$.

L'élimination conjointe des nœuds non décisifs et des arcs contigus correspond à ce qui est appelé « élimination des nœuds redondants » dans le contexte des BDDs booléens (les nœuds redondants sont ceux qui ont un seul fils).

Définition 4.1.12 (Nœud bégayant)**.** Un nœud non-racine N d'un IA φ est *bégayant* si et seulement si tous ses parents sont étiquetés par $\text{Var}(N)$ et soit $|\text{Out}(N)| = 1$, soit $|\text{In}(N)| = 1$.

Les nœuds bégayants sont inutiles, car l'information qu'ils apportent peut être reportée sur leurs parents (voir figure 4.5).

Définition 4.1.13 (Arc mort)**.** Un arc E d'un IA φ est *mort* si et seulement si $\text{Lbl}(E) \cap \text{Dom}(\text{Var}(E)) = \varnothing$.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

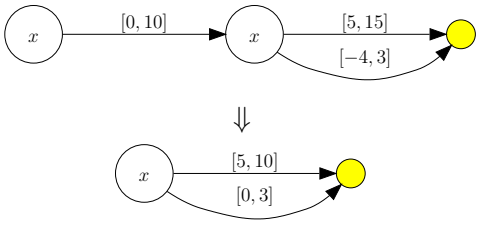


Fig. 4.5 : Fusion d'un nœud bégayant.

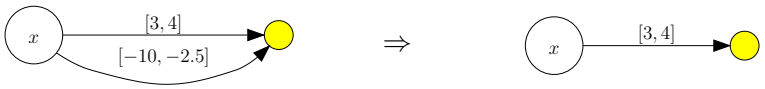


Fig. 4.6 : Élimination d'un arc mort (ici, $\mathrm{Dom}(x) = \mathbb{R}_+$).

Un arc mort n'est jamais franchi, puisque son étiquette ne contient aucune valeur qui soit cohérente avec le domaine de sa variable (voir figure 4.6).

Définition 4.1.14 (Automate réduit)**.** Un automate à intervalles φ est dit *réduit* si et seulement si :

- aucun nœud de φ n'est isomorphe à un autre, bégayant, ou non décisif ;
- aucun arc de φ n'est mort, ou contigu à un autre.

Dans la suite, on ne considérera toujours que des IAs réduits ; en effet la réduction est une opération polynomiale en la taille de la structure.

Proposition 4.1.15 (Réduction d'un IA)**.** Il existe un algorithme polynomial qui transforme un IA φ quelconque en un IA réduit équivalent φ' vérifiant $\|\varphi'\| \leqslant \|\varphi\|$.

Comme les BDDs booléens réduits, les IAs réduits ne sont *pas* canoniques : comme on le verra dans la carte de compilation, `IA` ne supporte pas la requête d'équivalence. Cependant, la réduction a des propriétés intéressantes ; elle élimine notamment les nœuds disjonctifs superflus, comme le montre la proposition suivante.

Proposition 4.1.16. Les seuls IAs réduits φ tels que $\mathrm{Scope}(\varphi) = \varnothing$ sont l'automate-puits et l'automate vide.

4.2 Un sous-langage efficace

Nous étudions à présent l'efficacité du langage `IA` pour les requêtes nécessaires à la planification [§ 3.3.2]. Cela nous conduira à proposer un sous-langage de `IA` ayant de meilleures performances.

4.2.1 Satisfaction d'importantes requêtes sur les automates à intervalles

La première opération que nous étudions est le *conditionnement*, qui consiste à assigner une valeur à certaines variables. Il est possible de faire ceci grâce à une simple procédure syntaxique ; grossièrement parlant, il s'agit de vérifier, pour chaque arc, si son étiquette contient la valeur à assigner ; les arcs satisfaisant cette condition sont gardés, les autres retirés.

Proposition 4.2.1. IA satisfait **CD**.

Le conditionnement se fait donc de manière assez directe sur les automates à intervalles. L'opération suivante dont nous avons besoin est la requête d'*extraction de modèle*. Malheureusement, cette dernière n'est pas supportée par IA — tout simplement parce qu'elle n'est pas supportée par $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$, et que $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$ est un sous-langage d'IA [corollaire 4.1.8].

Proposition 4.2.2. IA ne satisfait pas **MX**, sauf si $\mathsf{P} = \mathsf{NP}$.

Ceci est lié au fait que décider de la cohérence d'un automate à intervalles n'est pas polynomial. Une des raisons à cela est que les ensembles qui restreignent une variable le long d'un chemin peuvent se contredire : il est possible d'avoir un arc « $x \in [1, 4]$ » suivi d'un arc « $x \in [8, 10]$ », ce qui rend le chemin incohérent. Conséquemment, dans le pire des cas, il est nécessaire d'explorer tous les chemins avant d'un découvrir un qui soit cohérent. À noter que la réduction ne résout pas le problème ; la figure 4.7 montre ainsi un exemple d'IA réduit n'ayant aucun chemin cohérent.

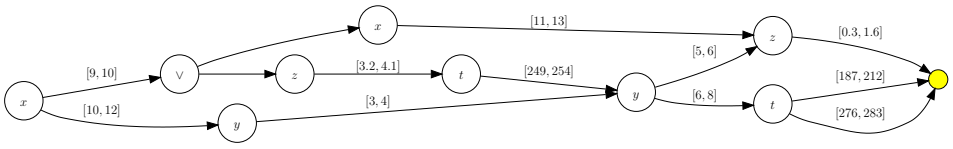


Fig. 4.7 : Exemple d'IA réduit dans lequel tous les chemins sont incohérents ; pour prouver son incohérence, il est nécessaire de vérifier chacun des chemins.

Étant donné que la requête **MX** est fondamentale dans nos applications, nous ne pouvons travailler avec les automates à intervalles de base ; nous devons imposer des restrictions structurelles rendant polynomiales la requête d'extraction de modèles et, par conséquent, la requête de cohérence.

4.2.2 Convergence

L'un des aspects rendant difficile la vérification de la cohérence d'un automate à intervalles est la potentielle incohérence des chemins, due au fait que les étiquettes portant sur une même variable peuvent être disjointes le long d'un chemin. Pour

définir un sous-langage d'`IA` interdisant ce comportement, la solution que nous proposons est de forcer les intervalles relatifs à un même variable à ne pouvoir que *rétrécir* de la racine au puits.

Définition 4.2.3 (Convergence)**.** Un arc E d'un IA φ est *convergent* si et seulement si tous les arcs E' le long de tout chemin de la racine de φ à $\mathrm{Src}(E)$ tels que $\mathrm{Var}(E) = \mathrm{Var}(E')$ vérifient $\mathrm{Lbl}(E) \subseteq \mathrm{Lbl}(E')$.

Un automate à intervalles est *convergent* (FIA, *focusing IA*) si et seulement si tous ses arcs E tels que $\mathrm{Var}(E) \neq \veebar$ sont convergents. `FIA` est la restriction de `IA` aux automates convergents.

Un exemple de FIA peut être trouvé en figure 4.8. Remarquons que la convergence n'est définie que sur les variables de $\mathrm{Scope}(\varphi)$: les nœuds $\veebar$ ne sont pas concernés par cette restriction. Cela permet de simplifier certaines preuves.

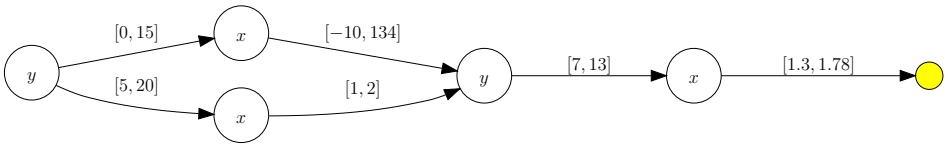


Fig. 4.8 : Un exemple d'automate à intervalles convergent. Les domaines des variables sont comme suit : $\mathrm{Dom}(x) = [0, 100]$, $\mathrm{Dom}(y) = [0, 100]$ et $\mathrm{Dom}(z) = \{0, 3, 7, 10\}$.

Imposer la convergence empêche les étiquettes de se contredire ; un chemin ne peut contenir « $x \in [1, 8]$ » puis « $x \in [5, 15]$ ». Cependant, la convergence seule ne rend pas tous les chemins cohérents : rien n'empêche les étiquettes des arcs d'être vides ou disjointes avec le domaine de leur variable. Ces cas sont en fait traités par la réduction ; c'est une des raisons pour lesquelles la polynomialité de la réduction sur les FIAs est un résultat important, qui fait l'objet de la proposition suivante.

Proposition 4.2.4 (Réduction d'un FIA)**.** Il existe un algorithme polynomial transformant un FIA φ quelconque en un FIA équivalent φ' réduit vérifiant $\|\varphi'\| \leqslant \|\varphi\|$.

On peut montrer un résultat intéressant à propos de la relation entre FIAs et diagrammes de décision *read-once*.

Proposition 4.2.5. On peut démontrer que $\mathtt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{FIA}$, et en particulier $\mathtt{FBDD}_{\mathcal{B}}^{\mathbb{SB}} \subseteq \mathtt{FIA}$.

Montrons à présent que les opérations qui nous intéressent sont polynomiales sur les FIAs, en commençant par le conditionnement. Pour prouver la proposition suivante, on utilise la procédure décrite sur les IAs — qui maintient la convergence.

Proposition 4.2.6 (Conditionnement d'un FIA)**.** `FIA` satisfait **CD**.

Avant de nous pencher sur l'extraction de modèle, remarquons que la réduction rend la vérification de cohérence polynomiale sur les FIAs. On peut en fait prouver qu'un FIA réduit est incohérent *si et seulement s'*il est vide : en effet, la seule chose

qui peut compromettre la cohérence d'un FIA est le fait qu'une étiquette « sorte » du domaine de sa variable, or cela est impossible sur un FIA réduit.

Proposition 4.2.7 (Cohérence d'un FIA)**.** `FIA` satisfait **CO**.

On peut voir cela d'une autre manière : chaque chemin d'un FIA réduit est cohérent, par conséquent, s'il existe un chemin, alors le FIA est cohérent. C'est cette propriété qu'on utilise pour extraire un modèle d'un FIA.

Proposition 4.2.8 (Extraction d'un modèle d'un FIA)**.** `FIA` satisfait **MX**.

La procédure est basée sur l'idée suivante : on réduit le FIA, on choisit un chemin arbitraire dans l'automate (le choix n'importe pas, puisque tous les chemins sont cohérents), puis on choisit une instanciation arbitraire comaptible avec ce chemin.

Ainsi, contrairement aux IAs de base, les FIAs supportent les principales opérations dont nous avons besoin pour des applications de planification. De plus, comme nous le montrons dans le chapitre 5, les automates à intervalles obtenus en suivant la *trace* d'un solveur de contraintes basé sur les intervalles sont naturellement convergents. Les FIAs sont donc de bons candidats pour être utilisés dans nos applications ; c'est pourquoi nos expérimentations ont porté sur ces structures (chapitre 6).

Avant de nous pencher sur la construction des FIAs, et d'expérimenter leur utilisation en pratique, nous présentons la carte de compilation de `IA` et `FIA`, contenant toutes les requêtes et transformations mentionnées dans les chapitres 1 et 3.

4.3 Carte de compilation d'`IA`

4.3.1 Préliminaires

Avant de présenter les résultats complets, nous détaillons quelques points-clefs sur lesquels la plupart des preuves reposent.

Contexte sur les FIAs

Pour extraire le contexte d'une variable dans un FIA, on utilise la propriété permettant de vérifier la cohérence et d'extraire un modèle, c'est-à-dire le fait que dans un FIA réduit, les étiquettes des arcs E tels qu'aucun autre arc relatif à la même variable n'est plus proche du puits que E, ne contiennent que des valeurs soit cohérentes soit extérieures au domaine de la variable.

Proposition 4.3.1 (Contexte dans un FIA)**.** `FIA` satisfait **CX**.

L'idée de la procédure est de trouver la x-frontière du puits, c'est-à-dire l'ensemble des nœuds N étiquetés par x tels qu'il existe un chemin d'un fils de N au puits ne mentionnant pas x ; en d'autres termes, les nœuds de la x-frontière sont les plus proches du puits. Si la racine est au-delà de la x-frontière, alors le contexte de x dans φ est $\mathrm{Dom}(x)$ (il existe un chemin de la racine au puits ne mentionnant pas x, et comme φ est réduit, ce chemin est trivialement cohérent). Sinon, le contexte de x est l'union des étiquettes des arcs par lesquels la x-frontière accède au puits, intersectée avec le domaine de x.

Implication clausale sur les FIAs

L'objectif de la requête d'implication clausale **CE** est de décider si une fonction booléenne donnée implique une clause donnée. Dans le cadre IA, cela consiste à vérifier si $\varphi \models \left[x_1 \in I_1\right] \vee \cdots \vee \left[x_k \in I_k\right]$, avec φ un IA, $x_1, \ldots x_k$ des variables, et $I_1, \ldots I_k$ des *intervalles fermés*. Cela revient à vérifier si $[\![\varphi]\!] \wedge \left[x_1 \notin I_1\right] \wedge \cdots \wedge \left[x_k \notin I_k\right]$ est incohérent. Étant donné qu'elle porte sur des intervalles ouverts, cette formule n'est, en toute généralité, pas représentable en automate à intervalles. Cependant, il est possible de construire en temps polynomial la restriction de φ au « terme » $\left[x_1 \notin I_1\right] \wedge \cdots \wedge \left[x_k \notin I_k\right]$, qui est cohérente si et seulement si la formule non représentable est cohérente.

Proposition 4.3.2 (Implication clausale dans un FIA)**.** FIA satisfait **CE**.

Combinaison d'automates

Remarquons que la construction d'une disjonction d'automates à intervalles est triviale, grâce aux nœuds disjonctifs.

Proposition 4.3.3 (Disjonction)**.** IA et FIA satisfont $\vee$**C**, $\vee$**BC** et $\vee$**clC**.

Construire une conjonction d'automates à intervalles est également simple, tant que l'on n'exige pas que l'automate résultant soit convergent ; l'idée est de connecter les automates les uns à la suite des autres.

Proposition 4.3.4 (Conjonction sur IA)**.** IA satisfait $\wedge$**C**, $\wedge$**BC** et $\wedge$**tC**.

Cette procédure ne maintient pas la convergence ; il est même impossible d'obtenir un FIA qui soit la conjonction de deux FIAs en temps polynomial.

Proposition 4.3.5 (Conjonction sur FIA)**.** FIA ne satisfait ni $\wedge$**BC** ni $\wedge$**C**, sauf si $\mathsf{P} = \mathsf{NP}$.

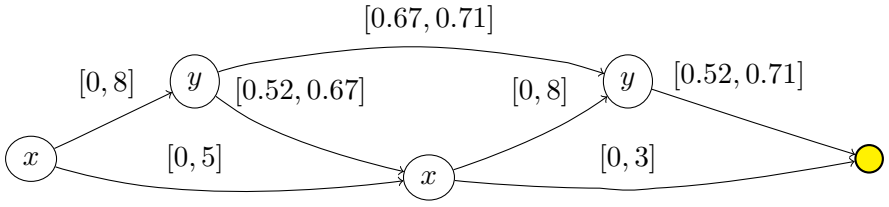
On peut néanmoins traduire un terme en FIA en temps polynomial.

Proposition 4.3.6 (Terme en FIA)**.** $\texttt{FIA} \leqslant_p \texttt{term}_{\mathcal{T}}^{\mathbb{IR}}$.

Maillages

Un certain nombre de preuves de polynomialité de requêtes et de transformations s'appuient sur le fait que les étiquettes de tous les arcs x dans un automate à intervalles induisent une *partition de* $\mathrm{Dom}(x)$, comme le montre la figure 4.9. Les éléments de cette partition ont la propriété d'être *plus fins* que n'importe quelle étiquette dans l'automate. On appelle une telle partition un *maillage*.

Définition 4.3.7 (Maillage d'une variable dans un IA)**.** Soit φ un IA et $x \in \mathrm{Scope}(\varphi)$. Un *maillage* de x dans φ est une partition $\mathcal{M} = \{M_1, \ldots, M_n\}$ de $\mathrm{Dom}(x)$ telle que pour tout arc E de φ vérifiant $\mathrm{Var}(E) = x$ et pour tout $i \in \{1, \ldots, n\}$, $(M_i \cap \mathrm{Lbl}(E) \neq \varnothing) \Rightarrow (M_i \subseteq \mathrm{Lbl}(E))$.



$$\big\{[0,3], \quad]3,5], \quad]5,8], \quad]8,10]\big\}$$
$$\big\{[0,0.52[, \quad [0.52,0.67[, \quad \{0.67\}, \quad]0.67,0.71], \quad]0.71,1]\big\}$$

Fig. 4.9 : Un IA sur les variables x et y de domaines respectifs $[0, 10]$ et $[0, 1]$. Les partitions (maillages) que cet IA induit sur ces domaines est montré en-dessous ; notons que chaque élément d'une de ces partitions est soit entièrement inclus, soit entièrement exclu de chacune des étiquettes dans l'IA.

Les maillages ont des propriétés intéressantes, provenant du fait que si $[a, b]$ est un des intervalles d'un maillage de x dans φ, alors conditionner φ en donnant à x n'importe quelle valeur dans $[a, b]$ donne toujours le même résultat. On peut qualifier les valeurs dans $[a, b]$ d'« interchangeables ». Cette propriété permet en particulier de quantifier (existentiellement ou universellement) les automates à intervalles en utilisant des opérations de conditionnement ; les preuves de **SFO** sur `IA` et `FIA` et de **SEN** sur `IA` utilisent ainsi ce mécanisme.

Proposition 4.3.8. Soit `L` un sous-langage de `IA` ; si `L` satisfait **CD** et $\vee$**C**, alors il satisfait **SFO**. Si `L` satisfait **CD** et $\wedge$**C**, alors il satisfait **SEN**.

4.3.2 Compacité

La relation de compacité relative entre la famille d'`IA` et les langages `CNF` et `DNF` est un résultat particulièrement important.

Proposition 4.3.9. Il est démontré que

$$\begin{matrix}\texttt{IA}\\ \texttt{FIA}\end{matrix} \quad \leqslant_p \quad \texttt{DNF}_{\mathcal{T}}^{\mathbb{IR}}$$

et

$$\texttt{IA} \quad \leqslant_p \quad \texttt{CNF}_{\mathcal{T}}^{\mathbb{IR}}.$$

Pour prouver des résultats négatifs sur la compacité, on se base sur un lemme fondamental, dû à Kautz et Selman [KS92a].

Lemme 4.3.10. Il est impossible de trouver une fonction de compilation comp en espace polynomial, telle que pour toute CNF propositionnelle Σ et toute clause propositionnelle γ, vérifier si $\Sigma \models \gamma$ en utilisant comp(Σ) puisse être fait en temps

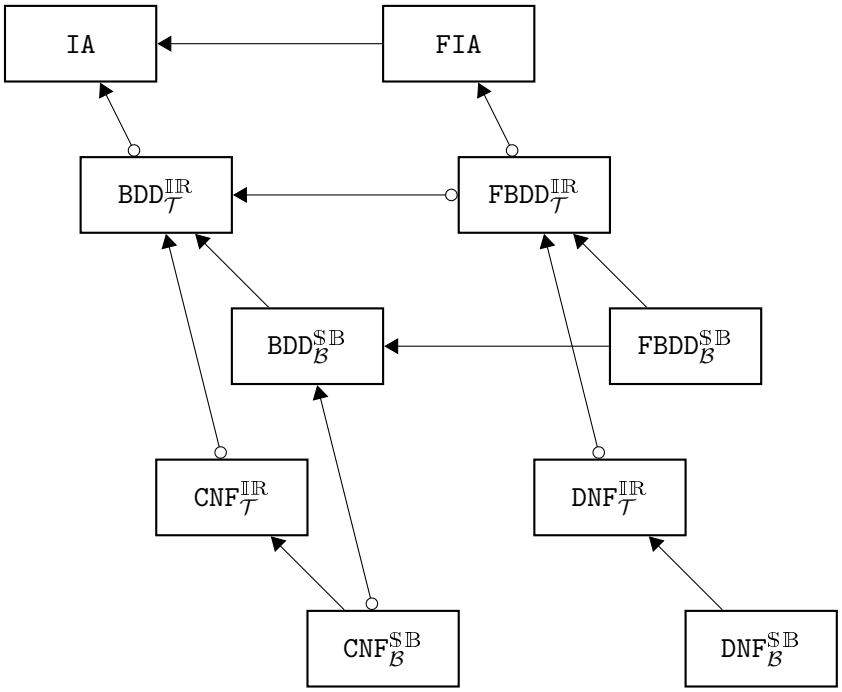


Fig. 4.10 : Compacité relative d'IA, FIA, et de divers langages sur des variables $\mathbb{R}$-pavables et booléennes. Sur un arc reliant $\mathtt{L}_1$ à $\mathtt{L}_2$, s'il y a une flèche pointant vers $\mathtt{L}_1$, cela signifie que $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$. S'il n'y a aucun symbole du côté de $\mathtt{L}_1$ (ni flèche, ni cercle), cela signifie que $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. Un cercle du côté de $\mathtt{L}_1$ signifie qu'on ignore si $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ ou si $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$. Les relations que l'on peut déduire par transitivité ne sont pas représentées, ce qui implique que deux fragments n'étant pas ancêtres l'un de l'autre sont incomparables sur le plan de la compacité.

polynomial, sauf si la hiérarchie polynomiale PH s'effondre au second niveau.

On utilisera surtout un corollaire de ce lemme, qui utilise une terminologie plus orientée « carte de compilation ».

Lemme 4.3.11. Si L est un langage de représentation booléen supportant **CE**, alors $\mathtt{L} \not\leqslant_s \mathtt{CNF}^{\$\mathbb{B}}_{\mathcal{B}}$, sauf si PH s'effondre au second niveau.

Ce lemme permet de prouver que FIA est strictement plus compact que IA, modulo l'effondrement de PH.

Proposition 4.3.12. $\mathtt{FIA} \not\leqslant_s \mathtt{IA}$, sauf si PH s'effondre.

Théorème 4.3.13. Les résultats présentés en figure 4.10 sont démontrés.

Démonstration. On le prouve directement à partir des propositions 4.2.5, 4.1.7, 4.3.9 et 4.3.12, et de diverses inclusions entre les langages. □

4.3.3 Requêtes et transformations

L	**CO**	**VA**	**MC**	**CE**	**IM**	**EQ**	**SE**	**MX**	**CX**	**CT**	**ME**
IA	○	○	√	○	○	○	○	○	○	○	○
FIA	√	○	√	√	○	○	○	√	√	○	√

Tab. 4.1 : Résultats sur les requêtes ; √ signifie « satisfait », et ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ ».

L	**CD**	**TR**	**FO**	**SFO**	**EN**	**SEN**	$\vee$**C**	$\vee$**BC**	$\vee$**dC**	$\wedge$**C**	$\wedge$**BC**	$\wedge$**tC**
IA	√	○	○	√	○	√	√	√	√	√	√	√
FIA	√	√	√	√	○	○	√	√	√	○	○	√

Tab. 4.2 : Résultats sur les transformations ; √ signifie « satisfait », et ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ ».

Théorème 4.3.14. Les résultats de compacité présentés dans les tableaux 4.1 et 4.2 sont démontrés.

Une grande partie des démonstrations provient directement de résultats de la carte de compilation existante [théorème 1.4.15]. Par exemple, le fait qu'IA ne satisfait aucune requête excepté le *model-checking* vient du fait que c'est le cas pour $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$; puisque $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}} \subseteq \mathtt{IA}$, IA ne peut supporter aucune requête que ne supporte pas $\mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}}$, comme l'indique la proposition 1.2.17.

Cela rend IA faible pour une vaste majorité des requêtes. En imposant la convergence, on rend polynomiales la plupart des requêtes, y compris **CO** et **MX**, qui sont particulièrement importantes dans notre contexte. Il est notable que malgré cette restriction, la plupart des transformations restent polynomiales, notamment l'oubli de variables. De ce point de vue, le « profil » (en termes de compilation de connaissances) de FIA est proche de celui de DNNF, FIA n'étant cependant pas un langage décomposable.

Notons que la transformation de négation ($\neg$**C**) n'apparaît pas dans la table. Cela est dû à l'expressivité d'IA, qui n'est pas fermée pour la négation, étant donné que seuls les intervalles fermés sont autorisés.

Remarquons en passant que le fait que les IAs et FIAs réduits ne soient pas canoniques — contrairement aux OBDDs — est un corollaire de ce théorème : en effet, s'ils étaient canoniques, **EQ** serait polynomiale.

*
**

Ayant présenté les aspects théoriques des automates à intervalles, nous passons dans le chapitre suivant au côté plus pratique de la *compilation* vers IA, et plus précisément vers FIA.

CHAPITRE

5

Construction d'automates à intervalles

Nous avons vu qu'en théorie, les automates à intervalles convergents permettaient de manipuler efficacement des variables à domaines continus ou énumérés. On peut donc adapter des approches existantes de « planification avec compilation » [§ 2.3], et en particulier celles que nous avons décidé d'étudier [§ 3.1], à l'utilisation de telles variables. Ainsi, il serait possible d'embarquer une politique de décision [§ 3.3.2.1] ou une table de transition [§ 3.3.2.2] pour permettre à un système autonome de prendre des décisions, sans devoir s'appuyer sur $\log_2(n)$ variables booléennes pour représenter chacune des variables continues dont on aurait discrétisé le domaine en n valeurs. Mais comment obtenir un FIA représentant une politique de décision ou une table de transition donnée ? Avant de les utiliser en pratique, nous devons étudier l'*étape de compilation* [§ 1.5.2].

À cette fin, nous nous penchons dans ce chapitre sur les diverses façons de construire des automates à intervalles convergents. Nous présentons tout d'abord le langage d'entrée, dans lequel les problèmes sont initialement représentés, celui des *réseaux de contraintes continus* [§ 5.1]. Nous détaillons ensuite la compilation *bottom-up* [§ 5.2] puis la compilation par trace d'un solveur [§ 5.3].

5.1 Réseaux de contraintes continus

Comme indiqué à la section 1.5.2, un moyen naturel de représenter les fonctions booléennes est le *réseau de contraintes* : un modèle d'une fonction booléenne est une *solution* du réseau de contraintes associé. Quand il porte sur des variables réelles, le problème de satisfaction de contraintes est généralement appelé *CSP continu* [SF96]. Nous utiliserons une terminologie similaire, bien que nous

n'autorisions pas n'importe quelle variable réelle, mais seulement celles qui sont $\mathbb{R}$-pavables.

Définition 5.1.1. Un *réseau de contraintes continu* (CCN, *continuous constraint network*) est un réseau de contraintes $\Pi = \langle V, \mathscr{C} \rangle$ tel que $V \subseteq \mathcal{T}$.

Transformer un CCN en FIA n'est pas une tâche triviale. Notons que la compilation en automate à intervalles comporte une importante différence avec la compilation en OBDD ; chaque contrainte d'un CN booléen est en effet représentable par un OBDD, ce qui permet notamment une compilation *bottom-up*, alors que l'incomplétude d'`IA` [proposition 4.1.6] ne le permet pas directement. Ainsi, des contraintes très simples telles que $x < 1$ ou $x = y$ ne peuvent être représentées exactement par des IAs.

La solution que nous avons retenue est l'utilisation d'un *solveur de contraintes basé sur les intervalles*, c'est-à-dire un outil conçu pour calculer une approximation de l'ensemble solution d'un CCN en divisant récursivement les domaines des variables en plusieurs morceaux. Nous avons utilisé le solveur RealPaver [GB06]. Sa sortie consiste en une liste de *boîtes*, qui sont directement compatibles avec le cadre des automates à intervalles.

RealPaver peut garantir qu'aucune solution n'est perdue, c'est-à-dire que toute solution du CCN est comprise dans au moins une des boîtes renvoyées. Sous certaines hypothèses, il est aussi capable de vérifier si l'ensemble de solutions est *exactement* l'union des boîtes renvoyées. Il utilise pour cela une distinction entre deux types de boîtes, les *extérieures*, qui peuvent potentiellement contenir des instanciations non solutions, et les *intérieures*, qui ne contiennent que des solutions. Nous avons décidé d'ignorer cette distinction, les automates à intervalles ne permettant de représenter qu'un seul type d'ensemble.

En résumé, l'incomplétude des automates à intervalles nous force à utiliser des approximations des fonctions booléennes que nous voulons compiler. Nous nous appuyons sur RealPaver pour calculer cette approximation, et nous contentons de compiler des automates à intervalles représentant les unions de boîtes renvoyées par RealPaver.

5.2 Compilation *bottom-up*

La compilation *bottom-up* est la méthode la plus directe [§ 1.5.2.1]. Nous étudions dans cette section son application aux automates à intervalles.

5.2.1 Union de boîtes

Le solveur basé sur les intervalles RealPaver approxime la fonction booléenne correspondant à un réseau de contraintes continu, en renvoyant une fonction booléenne qui, elle, est garantie d'être représentable par un IA. En effet, il renvoie une

liste de boîtes, et l'on sait que les fonctions représentables par des IAs sont précisément celles dont l'ensemble de solutions est une « union finie de boîtes ». Mais cela n'est pas seulement intéressant du point de vue de l'expressivité : les unions de boîtes sont de surcroît des $\mathtt{DNF}_{\mathcal{T}}^{\mathbb{IR}}$-représentations, que l'on sait être polynomialement traduisibles en `IA` et en `FIA` [proposition 4.3.9].

La première méthode que nous avons utilisée pour compiler un CCN en `FIA` consiste à résoudre ce CCN avec RealPaver, puis de traduire la sortie en FIA en effectuant la disjonction de toutes les boîtes renvoyées. Ici, RealPaver est simplement utilisé comme un outil pour approximer la fonction ; cette fonction approchée est alors compilée en FIA via une approche *bottom-up*.

Observons qu'il est également possible de compiler la sortie de RealPaver *à la volée*, c'est-à-dire en traduisant chaque boîte dès qu'elle est renvoyée et en ajoutant le FIA obtenu à la forme compilée courante. La compilation à la volée ne change ni la forme compilée finale, ni la complexité temporelle, mais évite d'avoir à garder en mémoire la sortie *complète* de RealPaver avant même de commencer la compilation.

5.2.2 Combinaison de contraintes

La méthode précédente pouvant être appliquée à des « réseaux » comportant une unique contrainte, il serait possible de compiler chaque contrainte, puis de combiner les automates élémentaires obtenus. Cette technique est utilisable pour la compilation d'IAs, mais pas de FIAs, étant donné que `FIA` ne satisfait pas la transformation $\wedge\mathbf{BC}$ — on peut potentiellement faire face à des pertes exponentielles en espace. Comme seul `FIA` supporte assez de requêtes pour nous être utile en ligne, nous devons écarter cette méthode.

5.3 RealPaver *with a trace*

Au lieu d'utiliser RealPaver comme une « boîte noire », en nous contentant de compiler sa sortie, nous pouvons adapter l'approche « DPLL *with a trace* » décrite en section 1.5.2.2. L'idée est de construire un IA correspondant exactement à la trace d'une recherche du solveur : chaque choix de variable crée un nœud, chaque partage de domaine crée un ensemble d'arcs. Nous commençons par examiner plus précisément l'algorithme de RealPaver.

5.3.1 Algorithme de recherche de RealPaver

L'algorithme 5.1 est une version simplifiée de l'algorithme générique *branch-and-prune* de RealPaver. Pour des raisons de clarté, nous le présentons sous une forme récursive, bien que l'implantation réelle soit itérative.

Son fonctionnement général est d'inspecter la boîte courante, d'en retirer les valeurs dont il peut prouver qu'elles ne correspondent à aucune solution (élagage),

puis de partager la boîte en plusieurs parties, et de s'appeler récursivement sur chacune des sous-boîtes ainsi obtenues. Il ne continue évidemment pas indéfiniment, mais s'arrête dès que la boîte courante est vide, ne contient que des solutions, ou est *assez petite*. Cette dernière condition est contrôlée par un paramètre de précision, donné par l'utilisateur.

Algorithme 5.1 `RealPaver`(Π, B, ε) : renvoie une union U de boîtes pas plus larges qu'ε, telle que U est incluse dans la boîte B, et que l'ensemble solution du CCN Π est inclus dans U.

```
1 : B^P ≔ Prune(Π, B)
2 : si B^P = ∅ alors
3 :     renvoyer ∅
4 : si B^P n'est pas incluse dans Sol(Π) alors
5 :     si B^P n'est pas plus précise qu'ε alors
6 :         y ≔ Sel_var(Π, B^P)
7 :         S ≔ Split(Π, B^P, y)
8 :         U ≔ ∅
9 :         pour tout B^s ∈ S faire
10 :            U ≔ U ∪ RealPaver(Π, B^s, ε)
11 :        renvoyer U
12 : renvoyer B^P
```

Plus précisément, la procédure prend en paramètre un CCN Π, duquel on note les variables $\{x_1, \ldots, x_n\}$ (les domaines des variables ne peuvent être que des intervalles, mais ce n'est pas un problème, voir section 5.3.5.3), une boîte $B = [a_1, b_1] \times \cdots \times [a_n, b_n]$, où chaque $[a_i, b_i]$ est un intervalle fermé inclus dans $\mathrm{Dom}(x_i)$, et un nombre positif ε contrôlant la précision.

C'est la fonction interne `Prune` qui est chargée de l'étape d'élagage ; elle utilise des techniques provenant de divers domaines [voir GB06 pour les détails]. Elle est également capable d'indiquer si la boîte élaguée B^P est ou non entièrement incluse dans l'ensemble solution du CCN ; cette information est utilisée ligne 4.

À la ligne 5, on compare la précision de la boîte élaguée B^P à la précision souhaitée. Ce test consiste à calculer la longueur de chaque $[a_i, b_i]$ dans la boîte élaguée : B^P est dite plus précise qu'ε si et seulement si *toutes* ces longueurs sont inférieures à ε.

L'étape de partage des domaines est contrôlée par deux fonctions internes, que nous appelons `Sel_var` et `Split`. La première se contente de choisir la prochaine variable sur laquelle brancher — cela peut dépendre d'une heuristique, mais c'est généralement la variable avec le plus grand domaine courant qui est choisie. La fonction `Split` divise ensuite la boîte en un certain nombre de boîtes plus petites qui la recouvrent (ce n'est pas une partition, car les boîtes sont des produits cartésiens d'intervalles *fermés*). Le nombre de sous-boîtes est de 2 ou 3, en fonction de la valeur d'un paramètre de RealPaver, mais cette valeur ne change jamais pendant la recherche.

5.3.2 Tracer RealPaver

On modifie RealPaver pour l'adapter aux principes décrits par Huang et Darwiche [HD05], destinés à compiler des structures booléennes en utilisant la trace de l'arbre de recherche d'un solveur. L'algorithme 5.2 est la réplique de l'algorithme 5.1, avec la différence qu'au lieu de renvoyer une union de boîtes, il renvoie un automate à intervalles. Les instructions utilisées dans ce but sont mises en évidence grâce à des cadres. Notons que cette procédure est une version simplifiée du véritable compilateur, qui a fait l'objet de modifications supplémentaires, et sera présenté dans la section suivante.

Algorithme 5.2 `RP_to_IA`(Π, B, ε) : renvoie un automate à intervalles représentant un surensemble du résultat de `RealPaver`(Π, B, ε).

```
1 :  B^P ≔ Prune(Π, B)
2 :  si B^P = ∅ alors
3 :      renvoyer l'automate vide
4 :  si B^P n'est pas incluse dans Sol(Π) alors
5 :      si B^P n'est pas plus précise qu'ε alors
6 :          y ≔ Sel_var(Π, B^P)
7 :          S ≔ Split(Π, B^P, y)
8 :          Ψ ≔ ∅
9 :          pour tout B^s ∈ S faire
10 :             soit ψ_s ≔ RP_to_IA(Π, B^s, ε)
11 :             Ψ ≔ Ψ ∪ { ⟨B^s_|y, ψ_s⟩ }
12 :         soit un nœud N ≔ Get_node(y, Ψ)
13 :         renvoyer l'IA ayant pour racine N
14 :     soit ψ l'automate-puits
15 :     renvoyer ψ
```

La procédure de compilation se base directement sur l'arbre de recherche : chaque nœud de l'arbre (c'est-à-dire chaque appel récursif à `RealPaver`) correspond à un nœud dans l'IA résultant. Les lignes 2 et 14 correspondent aux *feuilles* de l'arbre de recherche : dans le premier cas, on sait que la boîte courante ne contient aucune solution — on renvoie alors l'automate vide. Dans le second cas, la boîte courante n'est plus divisée, soit parce qu'elle ne contient que des solutions, soit parce qu'elle est suffisamment précise. On renvoie alors l'automate-puits.

Le cas restant est celui d'un nœud de recherche interne : la boîte courante est partagée, et la recherche continue pour chacune des sous-boîtes obtenues. Le partage d'une boîte en n parties selon une variable y revient à la création d'un nœud y avec n arcs sortants ; $B_{|y}$ est la *projection* de la boîte B selon la variable y — c'est donc un intervalle fermé.

La création d'un nœud interne dans l'automate est déléguée à une fonction

`Get_node`, qui prend en paramètre une variable x et un ensemble de n couples « intervalle–IA ». Elle construit un nœud étiqueté x avec n arcs sortants, chacun correspondant à l'un des couples : ainsi, chaque couple $\langle I, \varphi \rangle$ est associé à un arc sortant étiqueté I et pointant vers la racine de φ. L'avantage de la fonction `Get_node` est qu'elle peut être implantée de manière à effectuer certaines opérations de réduction à la volée. Par exemple, si pour l'un des couples $\langle I, \varphi \rangle$, $I = \varnothing$ ou φ est l'automate vide, l'arc correspondant n'est pas ajouté. S'il n'y aucun arc, aucun nœud n'est renvoyé. Il est également possible de vérifier si le nouveau nœud est non décisif — mais pas s'il est bégayant (car cette propriété dépend des nœuds parents). Cependant, `Get_node` peut vérifier si le nœud qu'elle crée est isomorphe à un autre nœud déjà existant, grâce à la technique de la *table de nœuds uniques* : chaque nœud existant est enregistré dans une table de hachage, avec une clé qui dépend de ses arcs sortants et de ses fils. Si le nœud créé par `Get_node` a la même clé qu'un nœud déjà présent dans la table de nœuds uniques, `Get_node` rejette le nouveau et renvoie à la place le nœud enregistré.

Notons qu'on ne crée *pas* de nœuds d'automate *à chaque fois* que la procédure ouvre un nouveau nœud dans l'arbre de recherche. En effet, `Get_node` ne peut créer que des « racines », elle ne peut construire d'arcs ne pointant vers aucune destination. Ainsi, elle n'est appelée que lors de la fermeture d'un nœud de recherche, c'est-à-dire quand le sous-arbre de recherche ayant pour racine le nœud de recherche courant a été entièrement exploré. Par conséquent, s'il s'avère après recherche que la boîte courante ne contient aucune solution, il n'y a aucun nœud à supprimer dans l'automate ; l'algorithme renvoie simplement l'automate vide.

L'automate courant n'est donc jamais plus grand que l'automate final. La table de nœuds uniques n'ayant besoin que d'une entrée par nœud de l'automate final, la quantité de mémoire nécessaire à l'algorithme est polynomiale en la taille de l'IA résultant. Remarquons cependant que cet IA résultant n'est *pas* réduit, puisque les nœuds bégayants ne peuvent être éliminés tant que l'automate n'est pas complet.

Nous avons pour l'instant présenté une adaptation directe de « DPLL *with a trace* », en ignorant l'étape d'élagage effectuée par RealPaver, qui n'a pas d'équivalent dans la version booléenne de Huang et Darwiche [HD05]. En effet, lorsqu'une valeur est retirée du domaine d'une variable booléenne, on ne branche plus jamais sur cette variable par la suite, puisqu'il ne reste aucune possibilité. Dans RealPaver, en revanche, un domaine peut être réduit plusieurs fois durant la recherche. Pour cette raison, les automates renvoyés par la procédure `RP_to_IA` ne représentent pas exactement l'union de boîtes renvoyée par RealPaver, mais seulement un surensemble de cette union. Par exemple, si on lui donne le CCN Π n'ayant qu'une seule variable x, de domaine $\mathbb{R}$, et une seule contrainte $x = 0$, la procédure `RealPaver` élague la boîte, la faisant passer de $\mathbb{R}$ tout entier au singleton $\{0\}$, se rend compte que la boîte obtenue est incluse dans l'ensemble de solutions de Π, et la renvoie. Pour cette entrée, `RP_to_IA` se contente de renvoyer l'automate-puits — duquel n'importe quelle $\{x\}$-instanciation est un modèle. Il est donc nécessaire de prendre en compte l'étape d'élagage pour que notre compilateur renvoie un IA équivalent à la sortie de RealPaver.

5.3.3 Prise en compte de l'élagage

Algorithme 5.3 `IA_builder`(Π, B, ε, L) : renvoie un automate à intervalles représentant l'ensemble calculé par `RealPaver`(Π, B, ε). L est l'ensemble des variables qu'il faut « restreindre » à la fin.

1 : $B^P \coloneqq$ `Prune`(Π, B)
2 : ajouter à L toutes les variables y vérifiant $B^P_{|y} \neq B_{|y}$
3 : **si** $B^P = \varnothing$ **alors**
4 : **renvoyer** l'automate vide
5 : **si** B^P n'est pas incluse dans Sol(Π) **alors**
6 : **si** B^P n'est pas plus précise qu'ε **alors**
7 : $y \coloneqq$ `Sel_var`(Π, B^P)
8 : $L \coloneqq L \setminus \{y\}$
9 : $S \coloneqq$ `Split`(Π, B^P, y)
10 : $\Psi \coloneqq \varnothing$
11 : **pour tout** $B^s \in S$ **faire**
12 : soit $\psi_s \coloneqq$ `IA_builder`$(\Pi, B^s, \varepsilon, L)$
13 : $\Psi \coloneqq \Psi \cup \left\{\langle B^s_{|y}, \psi_s\rangle\right\}$
14 : soit un nœud $N \coloneqq$ `Get_node`(y, Ψ)
15 : **renvoyer** l'IA ayant pour racine N
16 : soit ψ l'automate-puits
17 : **pour tout** $y \in L$ **faire**
18 : soit un nœud $N \coloneqq$ `Get_node` $\left(y, \left\{\langle B^P_{|y}, \psi\rangle\right\}\right)$
19 : soit ψ l'IA ayant pour racine N
20 : **renvoyer** ψ

Il est impossible de conserver une correspondance parfaite entre l'arbre de recherche de RealPaver et l'automate à intervalles résultant. En effet, l'étape d'élagage retire des valeurs de la boîte courante, et cela doit apparaître dans l'IA. Une première idée est de modifier l'IA renvoyé pour chaque boîte (aux lignes 13 et 15 de l'algorithme 5.2 [p. 105]), en ajoutant explicitement des nœuds restreignant le domaine de chaque variable aux valeurs non rejetées. En utilisant cette méthode, on multiplie donc le nombre de nœuds par un facteur égal au nombre de variables du CCN plus un (puisqu'on ajoute un nœud par variable et par nœud de l'automate).

Ajouter un nœud par variable n'est bien sûr pas nécessaire ; on n'a besoin de restreindre le domaine courant de la variable x que dans le cas où son domaine a changé lors de l'étape d'élagage. En procédant ainsi, on évite d'ajouter certains nœuds inutiles, mais pas tous ; ainsi, pour un chemin de l'arbre de recherche dans lequel une variable y a été modifiée k fois par la fonction `Prune`, on crée dans l'automate k « nœuds d'élagage », chacun d'entre eux étant inclus dans le précédent. Puisqu'ils n'ont tous qu'un seul arc sortant, on peut ne garder que le dernier sans

modifier l'ensemble de modèles du chemin.

Par conséquent, il semble plus adapté de n'ajouter des « nœuds d'élagage » qu'aux feuilles de l'arbre de recherche. Pour faire cela, il nous faut retenir durant la recherche les variables dont le domaine a été modifié au moins une fois par la fonction `Prune` ; on note L l'ensemble de ces variables. À chaque fois qu'une feuille de l'arbre de recherche est atteinte, au lieu de renvoyer l'automate-puits, la procédure renvoie l'IA représentant le terme $\bigwedge_{x \in L} \left[x \in B^P_{|x}\right]$, où B^P est la boîte élaguée courante. On garantit ainsi que l'élagage effectué sur chacun des domaines est reflété dans l'IA résultant, tout en limitant le nombre de nœuds ajoutés.

Enfin, on peut effectuer une dernière amélioration en remarquant que comme la fonction `Split` renvoie des sous-boîtes incluses dans la boîte élaguée courante, les modifications faites par `Prune` au domaine de la variable sur laquelle `Split` a branché sont déjà prises en compte. Si on note y cette variable, cela signifie donc qu'il n'est nécessaire d'ajouter un nœud d'élagage sur y uniquement si le domaine de y a été modifié *postérieurement* au dernier branchement sur y.

La procédure finale, appelée `IA_builder`, est décrite formellement dans l'algorithme 5.3. Les éléments nouveaux par rapport à l'algorithme 5.2 sont une nouvelle fois indiqués par des cadres. À la ligne 2, on met à jour la liste L des variables qui nécessitent un nœud d'élagage final ; à la ligne 8, on retire de L la variable sur laquelle on branche. Pour chaque feuille de l'arbre de recherche, on ajoute les nœuds d'élagage au-dessus de l'automate-puits grâce à la boucle de la ligne 17.

5.3.4 Exemple

Illustrons l'algorithme 5.3 sur un exemple. Soit Π le CCN ayant deux variables x et y, avec $\text{Dom}(x) = [0, 1]$ et $\text{Dom}(y) = [0, 2]$, et comportant les contraintes suivantes :

$$\begin{cases} y \geqslant 2\sqrt{\max(0, 0.25 - x^2)}, \\ y \leqslant 2 - 2\sqrt{\max(0, 0.25 - x^2)}, \\ y \geqslant 2\sqrt{\max(0, 0.25 - (x-1)^2)}, \\ y \leqslant 2 - 2\sqrt{\max(0, 0.25 - (x-1)^2)}. \end{cases}$$

La figure 5.1 propose une représentation graphique de l'ensemble de solutions de Π. L'union de boîtes renvoyée par RealPaver pour ce problème (avec $\varepsilon = 1$) est la suivante :

$$\begin{array}{rcrc} [0.5, 0.5669] \times [1.5, 2] & \cup & [0.5, 1] \times [1, 1.5] & \cup \\ [0.4330, 0.5] \times [1.5, 2] & \cup & [0, 0.5] \times [1, 1.5] & \cup \\ [0.5, 1] \times [0.5, 1] & \cup & [0.5, 0.5669] \times [0, 0.5] & \cup \\ [0, 0.5] \times [0.5, 1] & \cup & [0.4330, 0.5] \times [0, 0.5]. & \end{array}$$

La figure 5.1 montre l'IA correspondant obtenu grâce à l'algorithme 5.3.

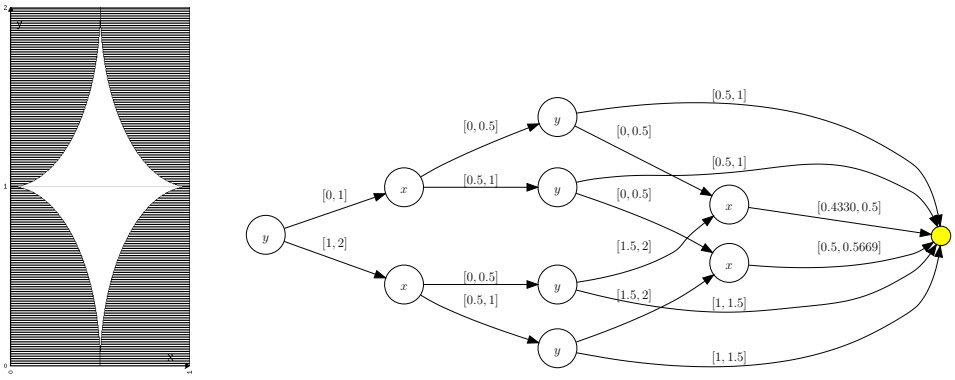


Fig. 5.1 : Cette figure illustre l'algorithme 5.3. La figure de gauche est une représentation graphique du problème Π défini dans le texte ; l'ensemble de solutions est l'aire non hachurée. L'IA de droite est le résultat donné par `IA_builder`.

5.3.5 Propriétés des IAs compilés

Structure

Les automates à intervalles obtenus grâce à la procédure `IA_builder` ont l'intéressante propriété d'être convergents. En effet, toutes les opérations faites par RealPaver qui mènent à la création d'un nœud dans l'automate, c'est-à-dire l'élagage et le partage des domaines, ne font que restreindre la boîte courante. Ainsi, lors de l'appel de `Get_node`, il est garanti que pour chacun des couples $\langle I, \varphi \rangle$, l'étiquette de chaque arc x dans φ est un sous-ensemble de I. `IA_builder` est donc un compilateur vers `FIA`, ce qui était un des objectifs nécessaires à son utilisation pour notre application.

Cohérence

Notons que si l'on obtient l'automate vide, cela montre que le problème était incohérent ; en effet, comme l'automate obtenu représente exactement la sortie de RealPaver, l'obtention d'un automate vide signifie que RealPaver a prouvé qu'il n'y avait aucune solution dans la boîte de départ.

Cependant, l'obtention d'un IA cohérent n'implique pas que le problème est cohérent : RealPaver est complet, mais pas correct, c'est-à-dire qu'il peut retourner des boîtes même si le problème est incohérent. Rappelons qu'à cause de l'incomplétude du langage `IA`, notre compilateur ne peut construire que des approximations de la véritable fonction booléenne.

Variables discrètes

Les variables du formalisme `IA` ont des domaines $\mathbb{R}$-pavables, ce qui signifie qu'elles peuvent être discrètes. Un IA peut donc représenter des fonctions booléennes dépendant à la fois de variables continues et discrètes. Cependant, RealPaver n'est capable de traiter que des variables ayant pour domaine des variables

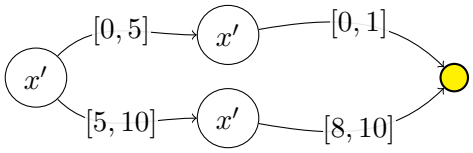


Fig. 5.2 : Le FIA obtenu avec `IA_builder` avec pour entrée le CCN d'une seule variable $x' \in [0, 10]$ et une seule contrainte $x' \notin]1, 8[$.

réelles[1]. Pour représenter une variable x de domaine $[0, 1] \cup [8, 10]$ par exemple, il est nécessaire de déclarer une variable x' de domaine $[0, 10]$, et d'ajouter une contrainte assurant que $x \notin]1, 8[$.

Détaillons comment RealPaver résoud le CCN trivial portant sur cette variable x et n'ayant aucune contrainte. Il est tout d'abord obligé de partager le domaine (si on suppose que la précision est inférieure à 10) : il crée deux boîtes, $[0, 5]$ et $[5, 10]$, puis élague chacune d'elles, obtenant $[0, 1]$ et $[8, 10]$. Le FIA obtenu est montré figure 5.2 : il contient 4 arcs (2 en éliminant le nœud bégayant), alors qu'il est possible de représenter cette fonction simplement par l'automate-puits. On améliore `IA_builder` en lui fournissant les *véritables* domaines des variables, et non simplement les informations données en entrée de RealPaver. En effet, si on remplace x' par x dans le FIA, il se réduit à l'automate-puits (par élimination du nœud bégayant, puis des arcs contigus, puis du nœud non décisif).

*
**

Nous avons implanté les deux méthodes de compilation décrites dans ce chapitre (union de boîtes et « RealPaver *with a trace* »). Dans le chapitre suivant, nous présentons des résultats expérimentaux sur la compilation de FIAs par ces méthodes, ainsi que sur leur efficacité opérationnelle.

[1] Il peut aussi prendre en entrée des variables ayant pour domaine des intervalles d'entiers, mais il ne s'agit que d'une commodité syntaxique ; pendant la recherche, ces variables se comportent exactement comme indiqué dans la suite de ce paragraphe.

CHAPITRE

6

Expérimentations sur les automates à intervalles

Nous avons vu dans le chapitre 4 que les automates à intervalles, en particulier ceux convergents, sont théoriquement bien adaptés aux applications de planification basées sur la compilation. Dans le chapitre 5, nous avons présenté des méthodes pour compiler des réseaux de contraintes continus en automates à intervalles convergents. Ce chapitre conclut la présente partie en résumant notre travail expérimental. Tout d'abord, nous présentons notre implémentation des automates à intervalles, des opérations sur `IA`, et de notre compilateur [§ 6.1]. Des résultats expérimentaux de compilation sont ensuite fournis [§ 6.2]. Pour finir, nous donnons des résultats concernant l'utilisation pratique des formes compilées [§ 6.3].

6.1 Implémentation

6.1.1 Cadre expérimental

Nous avons écrit une bibliothèque en Java dédiée à la manipulation des automates à intervalles. De la même façon que les paquetages classiques pour les BDDs, tels CUDD [Som05], notre programme permet la construction d'automates à intervalles par la méthode *bottom-up*, c'est-à-dire que les nœuds sont ajoutés incrémentalement, du puits à la racine, par une fonction `Get_node`, comme décrit dans la section 5.3.2. Presque toutes les opérations de réduction sont ainsi faites à la volée ; en particulier, la fusion des nœuds isomorphes s'appuie sur une table de nœuds uniques.

6.1.2 Compilateur d'automates à intervalles

Il est donc possible, avec notre bibliothèque, de construire les automates à intervalles « à la main », grâce à cette fonction `Get_node` ; mais cette approche étant assez limitée, nous avons implanté deux méthodes de compilation de réseaux de contraintes continus. Comme nous l'avons expliqué dans le chapitre précédent, ces deux techniques font usage du solveur RealPaver.

La première approche est la compilation de la sortie de RealPaver : nous avons écrit un programme qui analyse cette sortie et construit un FIA équivalent. En pratique, les deux étapes (résolution et compilation) sont séparées — mais rien n'empêcherait a priori de les entrelacer.

La seconde approche est « RealPaver *with a trace* » : nous avons implanté un prototype de ce compilateur, en modifiant RealPaver pour qu'il écrive la trace de sa recherche dans un fichier, et en utilisant ce fichier pour guider la compilation. Bien qu'il requière d'enregistrer la trace toute entière avant de commencer la compilation, notre prototype nous renseigne sur la faisabilité de l'approche, et sur les propriétés des IAs compilés en pratique.

6.1.3 Opérations sur les automates à intervalles

Nous avons implémenté un certain nombre d'opérations sur les IAs et les FIAs, en nous restreignant à celles ayant une complexité polynomiale. Les requêtes disponibles incluent la cohérence (**CO**) sur `FIA`, la vérification de modèle (**MC**), l'extraction de modèle (**MX**) sur `FIA`, et l'extraction de contexte (**CX**) sur `FIA`. Parmi les transformations disponibles se trouvent le conditionnement (**CD**), l'oubli (**FO**) sur `FIA`, la disjonction ($\vee$**C**), la conjonction ($\wedge$**C**) sur `IA`, et la conjonction avec un terme ($\wedge$**tC**).

Cependant, comme nous l'allons voir dans la section 6.3, ces opérations standard sont trop générales pour être efficaces en pratique. Les transformations, en particulier, sont souvent très gourmandes en temps et en espace. Nous avons pour cette raison également développé des opérations spécifiques à nos applications de planification. Les principales sont **CDCO**, **CDMX** et **CDFOMX**. Expliquons rapidement leur utilité et leur fonctionnement.

Pour le *benchmark Satellite*, nous avons besoin de conditionner la forme compilée en fonction de la situation courante, puis de vérifier si le FIA résultant est cohérent. Nous avons remarqué que cette tâche peut se voir comme une unique requête, au lieu de la combinaison de **CD** et **CO**. Cette requête « spéciale », que nous avons appelée **CDCO**, vise à répondre à la question « cette fonction est-elle cohérente une fois conditionnée par cette instanciation ? ». L'avantage d'utiliser notre requête spéciale est qu'il est possible d'obtenir la réponse par une simple traversée du graphe. L'idée est de maintenir un ensemble de nœuds « atteignables depuis la racine ». Initialement, cet ensemble ne contient que la racine ; on lui ajoute tous les fils de la racine qui sont accessibles modulo l'instanciation selon laquelle on veut conditionner ; on ajoute alors les enfants de ces nœuds, etc. Si, à un moment donné,

problème	durée (ms)	#nœuds	#arcs	taille carac.	fichier (octets)
*Drone*4-5-3	14 082	46 871	51 701	51 727	2 161 248
*Drone*4-10-3	20 317	97 722	104 576	104 602	4 237 494
*Drone*4-15-3	164 028	221 124	238 042	238 068	10 495 673
*Drone*4-20-3	146 504	223 984	245 078	245 104	10 868 910
*Drone*4-25-3	172 525	288 776	309 870	309 896	12 940 209
Satellite	25 391	141 096	145 016	145 059	8 476 904

Tab. 6.1 : Résultats obtenus en compilant la sortie de RealPaver en `FIA`.

le puits est ajouté à cet ensemble, on est sûr qu'il existe un modèle cohérent avec l'instanciation en question, et donc que la réponse à la requête est « oui ».

La deuxième requête spéciale est **CDMX** : son but est d'extraire un modèle d'un FIA conditionné, sans calculer explicitement ce conditionnement. Elle est notamment utile pour manipuler des politiques de décision (conditionnement sur l'état courant et extraction d'une décision) ainsi que des relations de transition (par exemple, conditionnement sur l'état et la décision choisie courants, et extraction d'un état suivant possible). Elle fonctionne de manière similaire à **CDCO** : schématiquement, on maintient un ensemble de nœuds atteignables depuis la racine, avec pour chacun l'arc entrant par lequel ils ont été atteints. Si le puits est ajouté, on peut récupérer un chemin cohérent en remontant le fil des arcs entrants jusqu'à la racine.

La dernière requête spéciale est une version modifiée de **CDMX**. Elle vise à extraire un modèle d'un FIA dans lequel certaines variables ont été conditionnées et d'autres oubliées. Nous l'appelons **CDFOMX** : la réponse peut être obtenue en appliquant **CD**, puis **FO**, puis **MX** — ou en appliquant la traversée de graphe décrite au paragraphe précédent, avec quelques modifications pour gérer les variables devant être oubliées.

6.2 Tests de compilation

Nous avons compilé, sur un ordinateur portable standard[1], diverses instances de nos problèmes portant sur des variables réelles, à savoir *Drone* et *Satellite* [voir chapitre 3], en utilisant les deux approches décrites au chapitre précédent. Pour chaque instance, nous donnons le temps nécessaire à la compilation du FIA (sans compter le temps d'exécution de RealPaver), le nombre de nœuds et d'arcs du FIA, sa taille caractéristique, et la taille en octets d'un fichier dans lequel le FIA a été écrit. Cette dernière taille est destinée à donner une vague idée de la taille mémoire relative des instances ; ce n'est en aucun cas une mesure précise de la taille qu'elles prennent dans la RAM lors de leur manipulation.

[1] Mobile Turion 64 X2 TL-56, 1.80 GHz, 2 Go RAM.

problème	durée (ms)	#nœuds	#arcs	taille carac.	fichier (octets)
*Drone*4-5-3	17 924	56 766	61 596	61 611	2 784 515
*Drone*4-10-3	21 723	74 436	81 290	81 299	3 718 465
*Drone*4-15-3	58 013	252 995	269 913	269 961	11 985 073
*Drone*4-20-3	92 206	329 724	350 818	350 836	15 595 018
*Drone*4-25-3	64 754	333 390	354 772	354 798	15 760 686
Satellite	24 899	109 078	112 022	112 045	6 546 529

Tab. 6.2 : Résultats obtenus avec « RealPaver *with a trace* ».

Le tableau 6.1 présente les résultats obtenus en compilant la sortie de RealPaver, et le tableau 6.2 présente les résultats de « RealPaver *with a trace* ». Cinq instances du *benchmark Drone* sont considérées, chacune avec quatre zones et trois billes disponibles, le paramètre qui varie étant le temps imparti.

Il est notable que les FIAs obtenus sont relativement gros, jusqu'à environ 300 000 nœuds et arcs. Utiliser des graphes d'une telle taille dans un système embarqué nécessite un espace mémoire non négligeable ; ce n'est cependant pas prohibitif pour les applications dans lesquelles la mémoire n'est pas une préoccupation cruciale. En revanche, la compilation est assez rapide. Pour les instances de *Drone*, il est plus rapide de compiler avec « RealPaver *with a trace* », mais cela donne des résultats de taille plus importante. Le fait que l'approche *bottom-up* nécessite plus de temps vient du fait que la clef de hachage de la table de nœuds uniques n'est pas très adaptée aux nœuds possédant de nombreux fils, ce qui est le cas de la racine (la forme compilée est en effet la disjonction de centaine de petits FIAs).

6.3 Tests applicatifs

Après avoir étudié les formes compilées au niveau de l'espace mémoire qu'elles occupent, nous étudions leur efficacité en termes de durée des opérations. Dans cette section, nous donnons des résultats de *simulations* de l'utilisation des formes compilées, c'est-à-dire que nous considérons quelques manières possibles de manipuler le FIA en ligne, en fonction du problème qu'il représente ; nous appelons *scénario* une utilisation possible de la forme compilée. Chaque scénario est divisé en étapes, qui correspondent à une requête ou une transformation. Nous donnons des résultats concernant l'exécution de ces scénarios sur les FIAs compilés avec « RealPaver *with a trace* ».

6.3.1 Simulation de l'utilisation de la relation de transition de *Drone*

Les instances du *benchmark Drone* sont des *relations de transition*, c'est-à-dire des relations liant un état courant et une action à l'état suivant correspondant : si

problème	**CD** $\vec{s}$	**FO** $\mathcal{S}'$	**MX** $\hookrightarrow\vec{a}$	**CD** $\vec{a}$	**MX** $\hookrightarrow\vec{s}'$	**CDFOMX** $\langle\vec{s},\mathcal{S}'\rangle\hookrightarrow\vec{a}$	**CDMX** $\vec{a}\hookrightarrow\vec{s}'$
*Drone*4-5-3	1074	6	0	2	0	0	1
*Drone*4-10-3	3145	18	0	3	0	0	0
*Drone*4-15-3	12 414	9	0	3	2	1	1
*Drone*4-20-3	14 936	4	1	5	0	1	1
*Drone*4-25-3	16 216	5	0	6	0	3	3

Tab. 6.3 : Résultats pour le scénario 1 sur plusieurs instances de la relation de transition de *Drone*, obtenue avec « RealPaver *with a trace* ». Toutes les durées sont en millisecondes.

on note $\mathcal{S}$ l'ensemble des variables d'état, $\mathcal{A}$ celui des variables d'action, et $\mathcal{S}'$ celui des variables d'état suivant, la forme compilée est une fonction booléenne sur $\mathcal{S} \cup \mathcal{A} \cup \mathcal{S}'$. Nous identifions quatre scénarios pouvant être utilisés sur de telles structures. Ils sont similaires, au sens où ils sont tous basés sur **CD**, **FO** et **MX**, mais les variables concernées et l'ordre des opérations diffèrent, ce qui a notamment un impact sur la taille de la forme compilée sur laquelle chaque opération est appliquée.

Dans le scénario 1, on observe un état courant ; on voudrait choisir une décision applicable, et obtenir un état suivant possible. Cela correspond aux opérations suivantes :

- **CD** sur le FIA tout entier, suivant une $\mathcal{S}$-instanciation $\vec{s}$;
- **FO** sur le FIA conditionné, suivant les variables de $\mathcal{S}'$;
- **MX** sur le FIA résultant, pour obtenir une action $\vec{a}$;
- **CD** sur le FIA conditionné, suivant $\vec{a}$;
- **MX** sur le FIA résultant, pour obtenir un état suivant $\vec{s}'$.

En utilisant les requêtes spéciales décrites dans la section 6.1.3, cela revient à

- **CDFOMX** sur le FIA tout entier, pour obtenir une action $\vec{a}$;
- **CDMX** sur le FIA tout entier, pour obtenir un état suivant $\vec{s}'$.

Nous avons lancé 20 simulations du scénario 1, en choisissant l'état courant initial au hasard parmi les états possibles. Le tableau 6.3 donne la durée moyenne de chaque opération, en millisecondes. Sans les requêtes spéciales, le goulet d'étranglement de ce scénario est le conditionnement. Cela vient du fait que de nombreux nœuds sont modifiés, or la modification d'un nœud a un impact sur tous ses ancêtres, à cause de la table de nœuds uniques. Une fois le premier conditionnement effectué, l'automate est beaucoup plus petit ; l'opération d'oubli et le deuxième conditionnement s'exécutent en un temps beaucoup plus raisonnable. Sans surprise, la requête d'extraction de modèle, qui *grosso modo* revient à une descente directe

problème	**CD** $\vec{s}$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}'$	**CDFOMX** $\langle \vec{s}, \mathcal{A} \rangle \hookrightarrow \vec{s}'$
*Drone*4-5-3	1074	14	0	0
*Drone*4-10-3	3145	40	2	0
*Drone*4-15-3	12 414	18	2	1
*Drone*4-20-3	14 936	7	0	1
*Drone*4-25-3	16 216	8	1	5

Tab. 6.4 : Résultats pour le scénario 2 sur plusieurs instances de la relation de transition de *Drone*, obtenue avec « RealPaver *with a trace* ». Toutes les durées sont en millisecondes.

dans le graphe, est très rapide — moins d'une milliseconde en moyenne pour toutes les instances.

En se limitant aux requêtes et transformations standard, le scénario entier nécessite 17 secondes en moyenne pour s'exécuter sur la plus grande instance. Cela peut être considéré comme acceptable pour certaines applications, en fonction notamment de la durée nécessaire pour effectuer chaque action. Mais si le drone doit prendre des décisions très rapidement, par exemple dans le cas où les zones sont très proches les unes des autres, la manipulation en ligne de cette table de transition à chaque prise de décision s'avère impossible. Cependant, grâce aux requêtes spécifiques **CDMX** et **CDFOMX**, le scénario entier peut être exécuté en 6 ms en moyenne pour les plus grosses instance, ce qui est bien plus raisonnable.

Dans le scénario 2, on observe également un état courant, mais on cherche cette fois un état suivant possible. Cela correspond aux opérations suivantes :

- **CD** sur le FIA tout entier, suivant une $\mathcal{S}$-instanciation $\vec{s}$;
- **FO** sur le FIA conditionné, suivant les variables de $\mathcal{A}$;
- **MX** sur le FIA résultant, pour obtenir un état suivant $\vec{s}'$.

Cela revient à une simple exécution de **CDFOMX**. Les résultats sont présentés dans le tableau 6.4 ; la première colonne contient les mêmes résultats que ceux du tableau 6.3, et l'opération en question (le conditionnement de la relation de transition entière) est une fois encore le facteur limitant du scénario. Oublier les variables d'action à la place des variables d'état suivant ne semble pas faire de différence significative : cela témoigne du fait que RealPaver ne suit aucun ordre de variables particulier.

Les scénarios 3 et 4 empruntent un chemin inverse, remontant dans le système états-transitions. Le but est de trouver un état (ou un couple état-action, respectivement) qui peut mener à l'état courant. La recherche d'un tel état peut être effectuée par les opérations suivantes :

- **CD** sur le FIA tout entier, suivant une $\mathcal{S}'$-instanciation $\vec{s}'$;

problème	**CD** $\overrightarrow{s}'$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow\overrightarrow{s}$	**CDFOMX** $\langle\overrightarrow{s}',\mathcal{A}\rangle$ $\hookrightarrow\overrightarrow{s}$	**CD** $\overrightarrow{s}'$	**MX** $\hookrightarrow\overrightarrow{s}.\overrightarrow{a}$	**CDMX** $\overrightarrow{s}'$ $\hookrightarrow\overrightarrow{s}.\overrightarrow{a}$
*Drone*4-5-3	1001	44	8	6	1001	9	8
*Drone*4-10-3	2778	29	4	15	2778	5	16
*Drone*4-15-3	11 590	1205	203	11	11 590	250	14
*Drone*4-20-3	18 700	1801	296	13	18 700	364	12
*Drone*4-25-3	19 865	1982	373	18	19 865	378	19

Tab. 6.5 : Résultats pour les scénarios 3 et 4 sur plusieurs instances de la relation de transition de *Drone*, obtenue avec « RealPaver *with a trace* ». Toutes les durées sont en millisecondes.

- **FO** sur le FIA conditionné, suivant les variables de $\mathcal{A}$;
- **MX** sur le FIA résultant, pour obtenir un état $\overrightarrow{s}$.

Cela correspond à une simple application de la requête spécifique **CDFOMX**. Pour trouver un couple état-action, il n'y a pas besoin d'oublier les variables d'action ; les opérations suivantes suffisent :

- **CD** sur le FIA tout entier, suivant une $\mathcal{S}'$-instanciation $\overrightarrow{s}'$;
- **MX** sur le FIA résultant, pour obtenir un couple état-action $\overrightarrow{s}\ .\ \overrightarrow{a}$.

Cela revient à exécuter **CDMX**. Les résultats pour les scénarios 3 et 4 sont montrés dans le tableau 6.5. Ils confirment nos observations précédentes, mis à part le fait que les opérations suivant le premier conditionnement prennent plus de temps que dans les scénarios « en avant ».

Pour résumer, les résultats montrent qu'il est a priori possible de manipuler en ligne des relations de transition compilées en FIAs, tant qu'on ne se contente pas de s'appuyer sur les opérations génériques standard.

6.3.2 Simulation de l'utilisation du sous-problème *Satellite*

Le *benchmark Satellite* représente une sous-partie d'un problème de prise de décision plus important. Pour utiliser la forme compilée, il suffit de la conditionner suivant l'état courant (c'est-à-dire l'attitude courante du satellite), et de vérifier si le résultat est cohérent. Si oui, cela signifie que la manœuvre de « pointage Soleil » est possible dans le temps imparti. Nous avons appliqué cet unique scénario sur la forme compilée du *benchmark Satellite* ; les opérations sont **CD** et **CO**, ce qui revient à la requête spécifique **CDCO**. Sur 20 états choisis au hasard, le conditionnement prend 13 173 ms et la vérification de cohérence moins de 1 ms en moyenne. Ceci est bien trop long pour cette application, dans laquelle le temps est un paramètre crucial. Cependant, notre requête spécifique **CDCO** prend également moins de 1 ms en

moyenne. Par conséquent, en supposant que la mémoire embarquée dans le satellite puisse contenir une forme compilée de cette taille, il est probable que la manipuler en ligne soit parfaitement envisageable.

Troisième partie

Set-labeled diagrams

Retour aux variables énumérées

Quelques remarques sur les maillages et la discrétisation

Comme GRDAG, le langage des automates à intervalles n'est pas complet [proposition 4.1.6] ; cela signifie qu'il existe des fonctions booléennes sur des variables $\mathbb{R}$-pavables qui ne peuvent être représentées par des IAs. En effet, chaque chemin dans un IA correspond à un terme $\left[x_1 \in A_1\right] \wedge \cdots \wedge \left[x_n \in A_n\right]$, dont l'ensemble de modèles est simplement une boîte $A_1 \times \cdots \times A_n$. Les automates à intervalles ne peuvent donc représenter que des « unions de boîtes », comme l'illustre la figure 6.1.

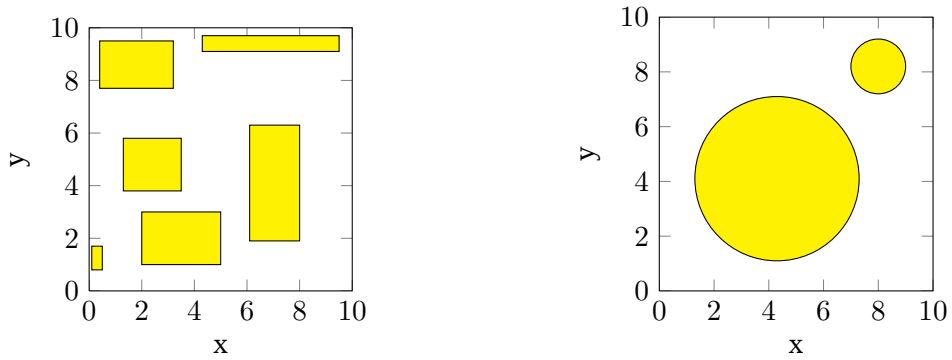


Fig. 6.1 : Un ensemble de modèles représentable par un IA (à gauche) et un autre non représentable par un IA (à droite).

La simple fonction $\left[x = y\right]$, dont plusieurs représentations sont montrées figure 6.2, par exemple, ne peut être exprimée de manière exacte par un IA : les automates étant finis, ils ne peuvent atteindre la précision nécessaire. On peut le voir sur la figure 6.2 ; deux valeurs « trop proches » l'une de l'autre dans le domaine d'une variable sont indiscernables dans l'IA.

Il est possible de « quantifier » la précision d'un IA donné. Au vu des grilles dessinées sur chaque ensemble de modèles de la figure 6.2, il apparaît clairement que l'IA de droite est beaucoup plus précis que celui de gauche. Cette grille correspond en fait à ce que nous avons appelé un *maillage* [définition 4.3.7], c'est-à-dire une partition de chaque domaine pour laquelle deux valeurs appartenant à un même morceau sont interchangeables. Pour un automate à intervalles donné, cette grille (et plus généralement n'importe quel maillage) constitue une *discrétisation* des domaines.

Ainsi, si l'on remplace chaque variable continue par une variable discrète — associant à chaque morceau du maillage une « méta-valeur » — et modifie les étiquettes des arcs pour refléter ce changement, on obtient un IA discret, qui représente une *discrétisation* de l'interprétation d'origine. L'intérêt de cette discrétisation repose dans le fait qu'elle n'a pas été choisie arbitrairement avant la construction

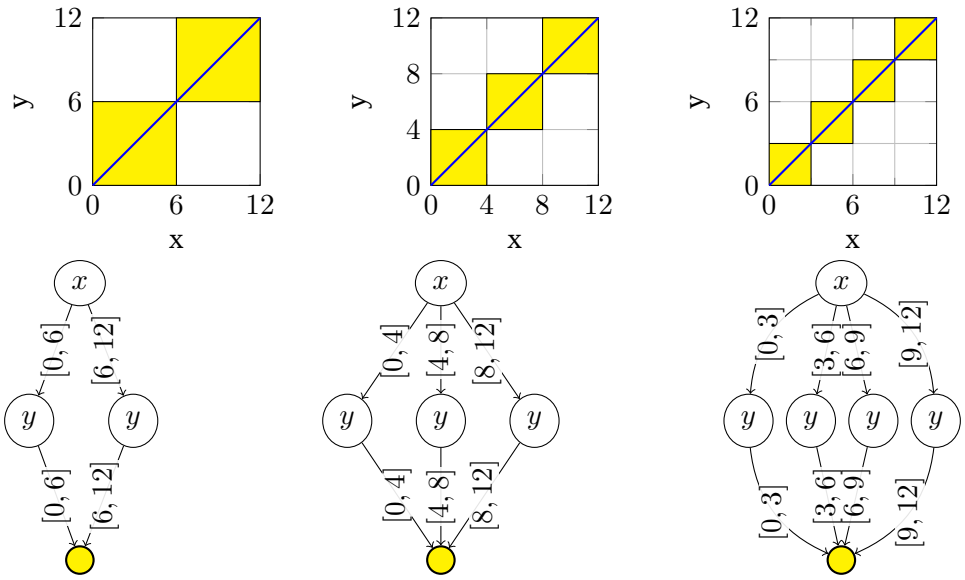


Fig. 6.2 : Trois IAs représentant la fonction booléenne $[x = y]$ avec une précision croissante.

de l'automate à intervalles, mais créée *a posteriori*, à partir de l'automate déjà construit. Cela permet en particulier une discrétisation plus adaptée à l'ensemble de modèles, la taille des éléments du maillage étant variable, qu'une discrétisation régulière arbitraire.

Cela est illustré figure 6.3 : la discrétisation arbitraire, à gauche, mène à des domaines énumérés de plus grande taille, sans pour autant être exacte, alors que la discrétisation de droite donne des domaines de taille minimale sans générer aucune approximation. Cela vient du fait que celle de droite a été calculée *après* la construction de l'IA.

Automates à intervalles discrets

En théorie, d'un point de vue orienté « carte de compilation », il est indifférent d'utiliser les automates à intervalles de base ou leur version discrétisée, la taille caractéristique des deux structures étant la même. Ce n'est cependant pas le cas en pratique, les nombres réels occupant plus d'espace que les entiers. Notre objectif étant d'embarquer des formes compilées, nous sommes très contraints par l'espace mémoire disponible, et l'utilisation d'IAs discrets paraît donc plus adaptée. Ils nécessiteraient bien sûr d'être embarqués avec une « table de traduction » fournissant les correspondances entre méta-valeurs discrètes et éléments continus des maillages, pour pouvoir coder et décoder les entrées et sorties des opérations.

Dans cette partie de la thèse, nous étudions un langage de représentation booléen constituant la version discrète du langage des automates à intervalles, en ce que la discrétisation sur maillage de tout IA est un élément de ce nouveau langage.

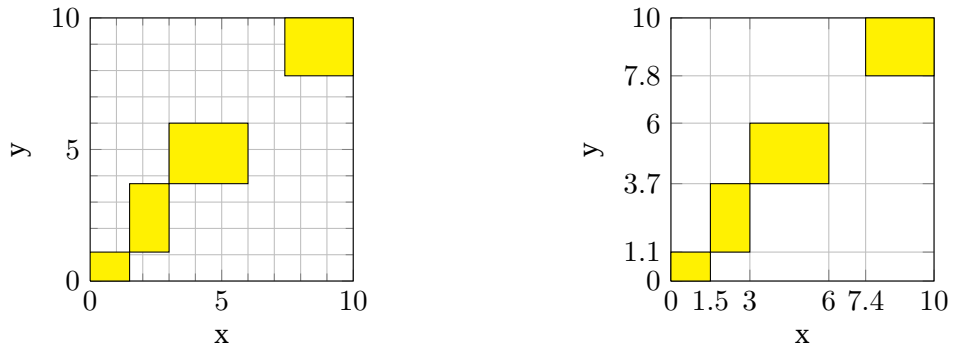


Fig. 6.3 : Un ensemble de modèles avec deux discrétisations différentes, représentées par des grilles. À gauche, une discrétisation arbitraire, approximative et coûteuse en espace mémoire. À droite, une discrétisation obtenue *a posteriori* à partir d'un IA compilé avec « RealPaver *with a trace* ».

Comme les valeurs dans les domaines des variables discrétisées n'ont pas d'importance en soi, nous choisissons de nous limiter à un cas simple, celui des variables portant sur un intervalle d'entiers. De plus, on augmente l'expressivité des arcs, qui pourront dorénavant être étiquetés par n'importe quel ensemble $\mathbb{Z}$-pavable. Cela ne change pas la taille caractéristique des représentations, mais permet, comme nous le verrons, d'augmenter la portée de la propriété de convergence.

Pour mettre l'accent sur la proximité de ces nouvelles structures avec la famille `BDD`, nous les appelons *set-labeled diagrams* (diagrammes à étiquettes ensemblistes). Cette dernière partie est consacrée à l'étude de ce langage et de ses sous-langages, et suit le même plan que la partie précédente : on commence, dans le chapitre 7, par définir formellement les langages en question et dresser leur carte de compilation, puis on examine des techniques pour les compiler dans le chapitre 8. Enfin, le chapitre 9 présente des résultats expérimentaux sur l'utilisation des *set-labeled diagrams* pour le contrôle des systèmes autonomes.

CHAPITRE

7

Famille des *set-labeled diagrams*

Dans ce chapitre, nous définissons le langage des *set-labeled diagrams*, qui sont des diagrammes de décision sur des variables énumérées. Nous montrons en quelle mesure ce langage est lié aux automates à intervalles ainsi qu'à la famille des diagrammes de décision de la littérature. Nous identifions des sous-langages ayant des propriétés intéressantes, en particulier celles de satisfaire les requêtes et transformations souvent utilisées dans le contexte de la planification.

Le plan sera le suivant : nous commençons avec des définitions portant sur les *set-labeled diagrams* [§ 7.1], puis introduisons divers sous-langages et évoquons leurs relations avec les langages existants [§ 7.2]. La section qui suit est dédiée à la relation spécifique liant les familles des automates à intervalles et des *set-labeled diagrams* [§ 7.3]. La dernière section contient la carte de compilation des *set-labeled diagrams* [§ 7.4].

7.1 Langage

7.1.1 Définition

Commençons par donner la définition formelle des variables que nous utiliserons dans ce cadre, les *variables entières*.

Définition 7.1.1 (Variables entières)**.** Une variable $x \in \mathcal{V}$ est une *variable entière* si et seulement si son domaine est un intervalle d'entiers : $\mathrm{Dom}(x) \in \mathbb{I}\mathbb{Z}$.

On ne considère que des variables entières ayant des valeurs contiguës, pour des raisons de simplicité ; cependant toute variable énumérée peut être représentée comme

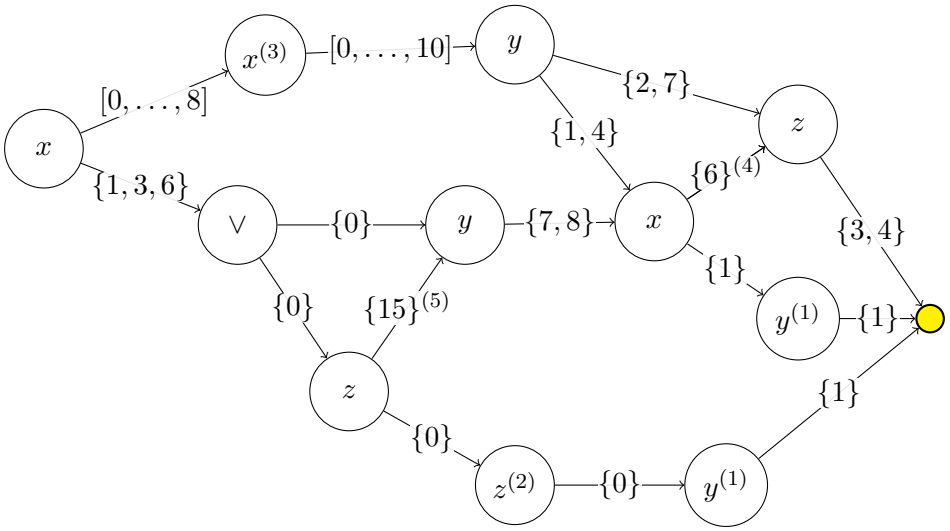


Fig. 7.1 : Un exemple de SD non réduit. Les domaines des variables sont tous $[0,\ldots,10]$. Les deux nœuds marqués $^{(1)}$ sont isomorphes ; le nœud $^{(2)}$ est bégayant ; le nœud $^{(3)}$ est non-décisif ; les arcs marqués $^{(4)}$ sont contigus ; l'arc $^{(5)}$ est mort.

une variable entière. Notons que le domaine d'une variable entière n'est pas nécessairement fini — $\mathbb{Z}$ tout entier est bien un intervalle fermé de $\mathbb{Z}$. Les éléments de $\mathbb{IZ}$ sont des ensembles de la forme $\{4,\ldots,9\}$. Pour souligner qu'il s'agit d'intervalles, nous utilisons la notation suivante : $[a,\ldots,b]$ pour $\{a,\ldots,b\}$, et $[a,\ldots,\infty]$ pour $\mathbb{N}\setminus\{0,\ldots,a-1\}$. Les points de suspension entre les bornes permettent de distinguer les intervalles réels des intervalles d'entiers. On note $\mathcal{I}$ l'ensemble des variables entières. Définissons à présent le langage des *set-labeled diagrams*, SD ; chaque partie de la définition formelle sera détaillé dans la suite.

Définition 7.1.2. Le langage SD est la restriction de $\mathtt{NNF}_{\mathcal{I}}^{\mathbb{TZ}}$ aux représentations satisfaisant la décision $\wedge$-simple [déf. 1.3.23].

L'ensemble $\mathbb{TZ}$ contient tous les ensembles $\mathbb{Z}$-pavables [définition 4.1.2], et la $\wedge$-simple décision [définition 1.3.23] signifie que les nœuds purement disjonctifs sont autorisés, mais pas les nœuds purement conjonctifs, exactement comme dans le cas des automates à intervalles. En d'autres termes, les SD-représentations sont des GRDAGs sur des variables entières, avec des littéraux de la forme « x appartient à une union d'intervalles », et une structure de type « diagramme de décision ». Cette définition est très proche de celle d'IA : les variables et l'expressivité littérale changent, mais la structure générale est la même. Pour cette raison, nous manipulerons les *set-labeled diagrams* (SDs) sous forme d'automates, comme nous l'avons fait pour les IAs [§ 4.1.2] — la seule différence étant les variables et les étiquettes d'arcs autorisées. La figure 7.1 donne un exemple de SD.

On utilise les notations introduites dans le contexte des IAs [§ 4.1.2] : $\mathrm{Var}(N)$ est l'étiquette d'un nœud N, etc. La seule différence est qu'ici, $\mathrm{Lbl}(E) \in \mathbb{TZ}$, alors

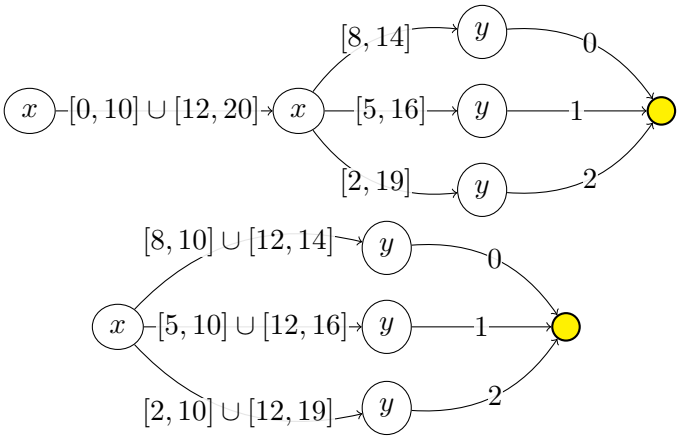


Fig. 7.2 : Illustration de la nouvelle définition du bégaiement. Fusionner des deux nœuds x dans le SD du haut donne le SD du bas, qui a une taille caractéristique supérieure. En conséquence, le nœud x du milieu dans le SD du haut ne peut être considéré comme bégayant.

que pour les automates à intervalles $\text{Lbl}(E) \in \mathbb{IR}$. Utiliser des ensembles pavables plutôt que des intervalles pour étiqueter les arcs permet de regrouper plus de valeurs : cela n'a aucun impact sur l'expressivité ou la taille caractéristique, mais est utile pour la convergence. En effet, intuitivement parlant, plus un arc sur x contient de valeurs, plus il est possible qu'il inclue toutes les étiquettes sur x qui suivent. Nous donnerons un exemple illustrant cet aspect un peu plus loin (figure 7.4).

7.1.2 Réduction

Il est possible de réduire les SDs en utilisant les même mécanismes que pour les IAs. Nous omettons la définition des notions de nœuds isomorphes, d'arcs morts et de nœuds non-décisifs, qui sont exactement les mêmes (définitions 4.1.9, 4.1.13 et 4.1.10) ; en revanche, les différences structurelles entre les SDs et les IAs (portée et expressivité littérale) nécessitent de redéfinir la contiguïté et le bégaiement.

Définition 7.1.3 (Arcs contigus)**.** Deux arcs E_1 et E_2 d'un SD φ sont *contigus* si et seulement si $\text{Src}(E_1) = \text{Src}(E_2)$ et $\text{Dest}(E_1) = \text{Dest}(E_2)$.

Cette définition est bien plus simple que celle sur les IAs, puisqu'il n'y a aucune condition sur les étiquettes. En effet, pour les SDs, l'union de deux étiquettes est *toujours* une étiquette valide, ce qui n'est pas le cas pour les IAs.

La définition d'un nœud bégayant est au contraire plus subtile (mais plus générale) pour les SDs que pour les IAs.

Définition 7.1.4 (Nœud bégayant)**.** Un nœud N non-racine d'un SD φ est *bégayant* si et seulement si tous les parents de N sont étiquetés par $\text{Var}(N)$, et de plus, soit $\sum_{E \in \text{Out}(N)} \|\text{Lbl}(E)\| = 1$, soit $\sum_{E \in \text{In}(N)} \|\text{Lbl}(E)\| = 1$.

Il ne suffit pas d'avoir $|\mathrm{Out}(N)| = 1$ ou $|\mathrm{In}(N)| = 1$. Si l'arc en question est étiqueté par une union, la procédure de réduction le retire, mais peut ce faisant augmenter la taille caractéristique des autres, comme montré sur la figure 7.2. C'est la raison pour laquelle il faut prendre en compte la taille caractéristique de l'étiquette : on décide qu'elle doit consister en un unique intervalle.

Définition 7.1.5 (Forme réduite)**.** Un SD φ est dit *réduit* si et seulement si aucun nœud de φ n'est isomorphe à un autre, bégayant ou non-décisif, et aucun arc de φ n'est mort ou contigu à un autre.

Comme pour les automates à intervalles, réduire un SD est une opération polynomiale en sa taille caractéristique.

Proposition 7.1.6 (Réduction)**.** Il existe un algorithme polynomial qui transforme tout SD φ en un SD réduit équivalent φ' vérifiant $\|\varphi'\| \leqslant \|\varphi\|$.

7.2 Sous-langages des *set-labeled diagrams*

7.2.1 La famille SD

Nous allons étudier un certain nombre de fragments de SD.

Définition 7.2.1 (Fragments de SD)**.** FSD est le fragment de SD satisfaisant la convergence [définition 4.2.3].

Soit $<$ un ordre strict total sur $\mathcal{I}$; $\mathtt{OSD}_<$ est la restriction de SD aux graphes ordonnés par $<$ [définition 1.3.25]. OSD est l'union de tous les $\mathtt{OSD}_<$, pour tout ordre strict total $<$.

SDD (resp. FSDD, $\mathtt{OSDD}_<$, OSDD) est la restriction de SD (resp. FSD, $\mathtt{OSD}_<$, OSD) aux propriétés de décision forte et exclusive [définition 1.3.23].

La notation « DD » se réfère aux *diagrammes de décision* ; on considère que satisfaire la décision forte et la décision exclusive — c'est-à-dire ne contenir que des nœuds purement disjonctifs ou conjonctifs, et que des nœuds de décision avec arcs sortants disjoints — est constitutif des diagrammes de décision.

Notons que, d'un point de vue formel, puisque $\veebar$ est une variable spéciale, elle n'est pas contrainte par la « décision exclusive ». De plus, cette variable spéciale visant à représenter une disjonction pure, un SD ne peut satisfaire la décision forte que s'il ne contient aucun nœud $\veebar$. Cependant, il faut remarquer qu'un nœud $\veebar$ satisfaisant la décision exclusive a au plus un seul arc sortant non mort ; il correspond donc à un nœud $\vee$ à un seul fils, qui peut être retiré sans conséquence. Formellement, l'exigence « décision forte et exclusive » sur les SDs au format GRDAG revient donc à une exigence plus simple sur les SDs au format diagramme de décision : « tous les nœuds (y compris $\veebar$) doivent satisfaire la décision exclusive ». Dans la suite, « décision exclusive » désignera toujours cette dernière version. Passons à présent à la hiérarchie des langages de la famille de SD ; on peut la représenter par un diagramme d'inclusion.

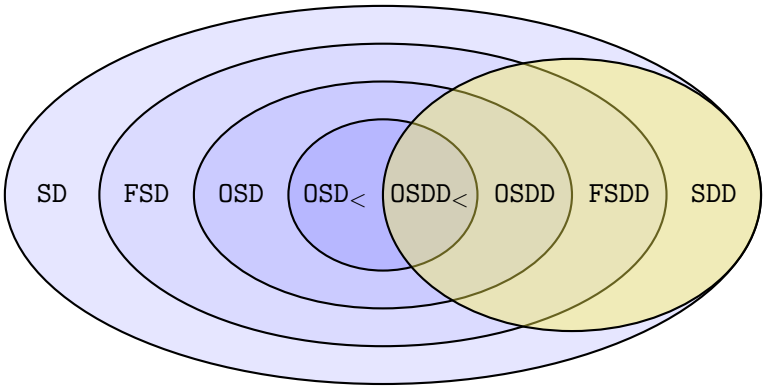


Fig. 7.3 : Hiérarchie d'inclusion des langages de la famille SD. Les langages satisfaisant la décision exclusive sont à l'intérieur du cercle (jaune) de droite.

Proposition 7.2.2. Les résultats d'inclusion présentés en figure 7.3 sont démontrés.

Pour tous ces fragments, la réduction fonctionne de la même manière que pour SD.

Proposition 7.2.3 (Réduction)**.** Soit L un des fragments de SD de la définition 7.2.1. Il existe un algorithme polynomial qui transforme n'importe quelle L-représentation φ en une L-représentation réduite équivalente φ' vérifiant $\|\varphi'\| \leqslant \|\varphi\|$.

7.2.2 Relations avec les familles IA et BDD

Notre objectif, en définissant les *set-labeled diagrams*, était de caractériser la forme prise par les automates à intervalles après une discrétisation suivant un maillage. Ces deux langages sont donc proches l'un de l'autre : cela est logique si l'on remarque qu'ils sont tous deux définis comme des sous-langages de NNF satisfaisant la décision $\wedge$-simple. Cependant, à cause de leurs domaines d'interprétation et expressivités littérales différents, aucun des deux n'est inclus dans l'autre.

Proposition 7.2.4. Il est démontré que $\mathtt{SD} \not\subseteq \mathtt{IA}$ et $\mathtt{IA} \not\subseteq \mathtt{SD}$.

Cependant, après restriction à des variables communes (c'est-à-dire à l'ensemble des variables entières à domaine fini, que nous avons appelé $\mathcal{E}$ en section 1.2.1) et des littéraux communs (l'ensemble des singletons d'entiers, $\mathbb{SZ}$), leur similarité de structure devient évidente.

Proposition 7.2.5. Il est démontré que $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}} = \mathtt{IA}_{\mathcal{E}}^{\mathbb{SZ}} = \mathtt{SD} \cap \mathtt{IA}$.

Notons que c'est également le cas pour les IAs et SDs convergents : $\mathtt{FSD}_{\mathcal{E}}^{\mathbb{SZ}} = \mathtt{FIA}_{\mathcal{E}}^{\mathbb{SZ}} = \mathtt{FSD} \cap \mathtt{FIA}$. En revanche, $\mathtt{SD}_{\mathcal{E}}^{\mathbb{SZ}}$ et $\mathtt{FSD}_{\mathcal{E}}^{\mathbb{SZ}}$ ne sont pas stables par la procédure de réduction que nous avons définie sur les SDs. La figure 7.4 montre pourquoi — et illustre au passage comment le choix d'étiqueter les arcs par des unions rend la convergence plus générale. Il est intéressant de constater, en passant, que restreindre SD aux variables énumérées rend le langage complet.

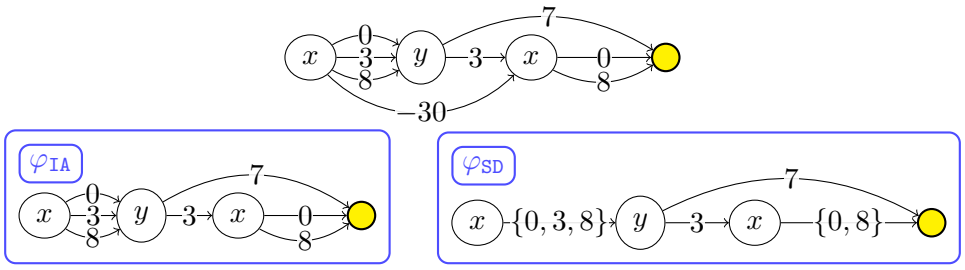


Fig. 7.4 : En haut : une $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}}$-représentation φ, sur les variables x et y de domaine $[0, \ldots, 10]$. En bas à gauche : la structure $\varphi_{\mathtt{IA}}$ obtenue en appliquant à φ la procédure de réduction de `IA`. En bas à droite : la structure $\varphi_{\mathtt{SD}}$ obtenue en appliquant à φ la procédure de réduction de `SD`. On voit que $\varphi_{\mathtt{IA}}$ est toujours dans $\mathtt{SD}_{\mathcal{E}}^{\$\mathbb{Z}}$, alors que ce n'est pas le cas de $\varphi_{\mathtt{SD}}$. En revanche, $\varphi_{\mathtt{SD}}$ est convergent, quand $\varphi_{\mathtt{IA}}$ ne l'est pas.

Proposition 7.2.6. `SD` est incomplet, mais $\mathtt{SD}_{\mathcal{E}}$ est complet.

Regardons à présent quelles relations entretiennent les familles `SD` et `BDD`. Étant donné que les SDs sont proches des automates à intervalles, et que l'on sait que $\mathtt{BDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{IA}$ et $\mathtt{FBDD}_{\mathcal{T}}^{\mathbb{IR}} \subseteq \mathtt{FIA}$, on s'attend à des résultats similaires pour les SDs.

Proposition 7.2.7. Les propriétés suivantes sont démontrées :

$$\begin{aligned}
\mathtt{BDD}_{\mathcal{I}}^{\mathbb{TZ}} &= \mathtt{SDD},\\
\mathtt{FBDD}_{\mathcal{I}}^{\mathbb{TZ}} &\subsetneq \mathtt{FSDD},\\
\mathtt{OBDD}_{\mathcal{I}}^{\mathbb{TZ}} &= \mathtt{OSDD},\\
\mathtt{OBDD}_{<,\mathcal{I}}^{\mathbb{TZ}} &= \mathtt{OSDD}_{<}.
\end{aligned}$$

Cette proposition est une des raisons pour lesquelles nous avons utilisé « DD » dans le nom des SDs satisfaisant la décision exclusive : `SDD` correspond exactement au langage général `BDD` restreint aux variables entières et aux littéraux $\mathbb{Z}$-pavable. En particulier, `SDD` contient les BDDs booléens classiques, comme le montre la proposition suivante.

Proposition 7.2.8. Il est démontré que

$$\begin{aligned}
\mathtt{SDD}_{\mathcal{B}}^{\$\mathbb{B}} &= \mathtt{BDD}_{\mathcal{B}}^{\$\mathbb{B}},\\
\mathtt{OSDD}_{\mathcal{B}}^{\$\mathbb{B}} &= \mathtt{OBDD}_{\mathcal{B}}^{\$\mathbb{B}},\\
\mathtt{OSDD}_{<,\mathcal{B}}^{\$\mathbb{B}} &= \mathtt{OBDD}_{<,\mathcal{B}}^{\$\mathbb{B}}.
\end{aligned}$$

On peut également montrer que $\mathtt{MDD} = \mathtt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$, comme $\mathtt{MDD} = \mathtt{OBDD}_{\mathcal{E}}^{\$\mathbb{Z}}$. Cependant, il est notable que $\mathtt{MDD} \subsetneq \mathtt{OSDD}_{\mathcal{E}}$, à cause de l'expressivité littérale de `OSDD`. On peut même facilement montrer que $\mathtt{OSDD}_{\mathcal{E}}$ est strictement plus succinct que `MDD`.

Proposition 7.2.9. Il est démontré que $\texttt{MDD} \not\leq_s \texttt{OSDD}_{\mathcal{E}}$.

En résumé, la famille SD est très liée aux autres langages de type « diagrammes de décision », mais généralise les diagrammes de décision binaires et multivalués selon plusieurs axes :

- variables plus générales que $\mathcal{B}$ et $\mathcal{E}$;
- expressivité littérale moins contrainte, ce qui permet des gains en espace potentiellement exponentiels ;
- relâchement de la décision exclusive sur certains fragments, avec notamment la possibilité d'utiliser des nœuds purement disjonctifs.

7.3 Des IAs aux SDs

Dans la section précédente, on a vu que les automates à intervalles ne sont pas des *set-labeled diagrams* particuliers, et *vice-versa*. Il est malgré tout possible de transformer les IAs en SDs en les discrétisant, ainsi que nous l'avons expliqué dans l'introduction de la partie. Dans cette section, nous présentons cette discrétisation de manière formelle, et démontrons l'« équivalence » de l'IA et de sa discrétisation. Étant donné que IA et SD n'ont pas le même domaine d'interprétation, nous ne pouvons prouver cette équivalence dans notre cadre formel ; nous devons au préalable décider ce que cette « équivalence » signifie.

7.3.1 Une définition formelle de la discrétisation

Discrétiser une fonction continue consiste à trouver une fonction discrète ayant le « même comportement ». Une discrétisation est souvent une approximation de la fonction originale. Néanmoins, les fonctions représentées par des IAs ont une propriété spécifique, que nous avons soulignée dans l'introduction de cette partie : leur ensemble de modèles est toujours une union finie de boîtes. Cela rend possible une discrétisation exacte, comme nous l'allons voir.

Nous devons en premier lieu définir formellement le concept de discrétisation d'un variable. Il s'appuie sur une partition de son domaine, et consiste en une variable discrète munie de fonctions associant les valeurs continues aux valeurs discrètes.

Définition 7.3.1 (Discrétisation d'une variable)**.** Soit x une variable $\mathbb{R}$-pavable, et p une partition *finie* de $\mathrm{Dom}(x)$. La *discrétisation de x selon p* est le triplet $\delta = \langle d, \sigma, \pi \rangle$, où

- $d \in \mathcal{E}$ est une variable énumérée, dont le domaine vérifie $|\mathrm{Dom}(d)| = |p|$;
- σ est une bijection de p sur $\mathrm{Dom}(d)$;

- $\pi\colon \mathrm{Dom}(x) \to p$ est la *fonction de partition*, associant chaque valeur $\omega \in \mathrm{Dom}(x)$ à l'unique élément de p contenant ω.

Avec cette définition, chaque valeur ω de $\mathrm{Dom}(x)$ correspond à une unique valeur ω_δ de $\mathrm{Dom}(d)$, donnée par

$$\omega_\delta = \sigma(\pi(\omega)),$$

et chaque valeur discrète ω_δ de $\mathrm{Dom}(d)$ correspond à un ensemble de valeurs « interchangeables » ω de $\mathrm{Dom}(x)$, vérifiant

$$\omega \in \sigma^{-1}(\omega_\delta).$$

À noter que l'on peut trouver une discrétisation triviale pour toute variable $\mathbb{R}$-pavable x : en effet, $\{\mathrm{Dom}(x)\}$ est une partition finie de $\mathrm{Dom}(x)$.

On peut étendre cette définition à un ensemble de variables, en joignant les fonctions σ et π de chaque variable.

Définition 7.3.2 (Discrétisation d'une portée)**.** Soit $X \subseteq \mathcal{T}$ un ensemble fini de variables $\mathbb{R}$-pavables, noté $X = \{x_1, \ldots, x_n\}$. Pour chaque $x_i \in X$, on considère une partition finie p_i de $\mathrm{Dom}(x_i)$, et δ_i la discrétisation de x_i selon p_i : $\delta_i = \langle d_i, \sigma_i, \pi_i \rangle$.

Une *discrétisation de* X est un triplet $\Delta = \langle D, \Sigma, \Pi \rangle$, avec

- $D = \{d_1, \ldots, d_n\}$ l'ensemble de tous les d_i ;
- $\Sigma\colon p_1 \times \cdots \times p_n \to \mathrm{Dom}(D)$, la fonction définie comme la concaténation des fonctions $\sigma_1, \ldots, \sigma_n$;
- $\Pi\colon \mathrm{Dom}(X) \to p_1 \times \cdots \times p_n$ la fonction de partition, définie comme $\Pi(\vec{x}) = (\pi_1(\vec{x}_{|x_1}), \ldots, \pi_n(\vec{x}_{|x_n}))$.

De la même manière que pour les variables, chaque X-instanciation $\vec{x}$ correspond à une unique D-instanciation $\vec{d} = \Sigma(\Pi(\vec{x}))$, et une D-instanciation $\vec{d}$ donnée est associée à un ensemble $\Sigma^{-1}(\vec{d})$ de X-instanciations.

Nous avons maintenant tous les éléments nécessaires pour définir la notion de discrétisation d'une fonction booléenne. Étant donné que nous voulons des discrétisations exactes, la valeur de la fonction originale pour une instanciation donnée doit être la même que celle de la fonction discrète pour l'instanciation discrète correspondante.

Définition 7.3.3 (Discrétisation d'une fonction)**.** Soit $f \in \mathfrak{D}_{\mathcal{T},\mathbb{B}}$ une fonction booléenne sur des variables $\mathbb{R}$-pavables, et X sa portée. Soit $\Delta = \langle D, \Sigma, \Pi \rangle$ une discrétisation de X. Une *discrétisation de f selon Δ* est une fonction booléenne $f_\Delta \in \mathfrak{D}_{\mathcal{E},\mathbb{B}}$, de portée D, telle que

$$f \equiv f_\Delta \circ \Sigma \circ \Pi.$$

Il n'existe pas toujours de discrétisation pour une fonction donnée. En effet, on exige que la discrétisation soit *équivalente* à la fonction continue. Par consé-

quent, puisque les partitions des domaines doivent être finies, les fonctions non-représentables comme des unions de boîtes n'ont aucune discrétisation. Considérons une fois encore la fonction $\llbracket x = y \rrbracket$, par exemple : pour qu'elle ait une discrétisation équivalente, il faudrait des partitions de $\mathrm{Dom}(x)$ et $\mathrm{Dom}(y)$ de taille infinie (par exemple $\mathbb{SR}$). En n'utilisant que des partitions finies, il est toujours possible de trouver deux $\{x, y\}$-instanciations donnant des résultats différents par f, mais correspondant pourtant à la même $\{d_x, d_y\}$-instanciation, et conduisant donc au même résultat par f_Δ.

En revanche, il existe une discrétisation pour toute fonction représentable par un IA, ou, de manière équivalente, $\mathrm{Expr}(\texttt{IA})$ [définition 1.2.10] est inclus dans l'ensemble des fonctions discrétisables. Il s'agit en fait d'un corollaire du fait que tout IA peut être discrétisé — que nous montrons dans la section suivante.

7.3.2 Transformer des IAs en SDs

L'algorithme 7.1 implante la procédure que nous avons informellement décrite en introduction de la partie. Il construit un SD qui correspond à l'IA d'entrée, en utilisant les maillages obtenus en listant les bornes de tous les intervalles rencontrés, et des *suites d'indexation* — c'est-à-dire, pour chaque maillage $\mathcal{M} = \{M_1, \ldots, M_n\}$, une suite de valeurs $\mathrm{Indexes} = (m_1, \ldots, m_n)$ telle que $m_i \in M_i$ pour tout i entre 1 et n. Le SD obtenu a la même taille caractéristique que l'IA d'entrée (voire même une taille inférieure, puisque les domaines des variables, qui sont des intervalles, ont une taille caractéristique de 1 dans le cadre $\texttt{SD}$), et est compatible avec notre définition formelle d'une discrétisation. Nous utilisons cette procédure pour prouver la proposition suivante.

Algorithme 7.1 Étant donné un IA φ, et une suite d'indexation $\mathrm{Indexes}^i$ pour chaque variable x_i dans $\mathrm{Scope}(\varphi)$, construit un SD ψ représentant une discrétisation de $[\![\varphi]\!]$.

soit ψ le SD vide
pour tout nœud N de φ, ordonnés du puits à la racine **faire**
    soit $x_i \coloneqq \mathrm{Var}(N)$
    ajouter à ψ un nœud N' étiqueté d_i
    pour tout $E \in \mathrm{Out}(N)$ **faire**
        soit $S \coloneqq \varnothing$
        pour tout $m_j \in \mathrm{Indexes}^i$ **faire**
            si $m_j \in \mathrm{Lbl}(E)$ **alors**
                $S \coloneqq S \cup \{j\}$
        soit N'_E le nœud de ψ correspondant à $\mathrm{Dest}(E)$
        ajouter à ψ un arc de N' vers N'_E, étiqueté S

Proposition 7.3.4. Il existe un algorithme polynomial associant tout IA φ à un SD ψ, de taille $\|\psi\| \leqslant \|\varphi\|$, et tel que $[\![\psi]\!]$ est une discrétisation de $[\![\varphi]\!]$.

Cela signifie qu'en utilisant cette discrétisation, on ne perd ni espace mémoire ni information. Embarquer un SD au lieu d'un IA (avec des tables pour représenter chaque bijection σ_i) est donc parfaitement équivalent d'un point de vue sémantique, et ne peut que permettre de gagner de l'espace : il n'est nécessaire d'embarquer que $\sum_{x_i \in \text{Scope}(\varphi)} 2n_i$ nombres réels, pour représenter les bijections σ (deux nombres réels pour chaque intervalle de chaque maillage). Par construction d'un maillage, cela représente forcément moins de nombres réels que ceux nécessaires à représenter φ — et potentiellement beaucoup moins, par exemple lorsqu'une même borne est utilisée plusieurs fois dans le graphe.

Cela serait évidemment inutile si la procédure ne maintenait pas la convergence — mais c'est bien le cas.

Proposition 7.3.5. Il existe un algorithme polynomial associant à tout FIA φ un FSD ψ, de taille $\|\psi\| \leqslant \|\varphi\|$, et tel que $[\![\psi]\!]$ est une discrétisation de $[\![\varphi]\!]$.

Cette section visait à fournir une motivation formelle à l'étude de la famille SD. À présent que nous savons que les FIAs peuvent être représentés efficacement par des FSDs, nous allons étudier les propriétés des langages de cette famille d'un point de vue « compilation de connaissances ».

7.4 Carte de compilation de la famille SD

Cette section contient la carte de compilation de la famille SD. Commençons par présenter quelques propriétés fondamentales, qui sont à l'origine d'un grand nombre de résultats de la carte.

7.4.1 Préliminaires

Combiner des SDs

De la même façon que pour les automates à intervalles et les diagrammes de décision binaires, il existe des procédures simples pour combiner des SDs, en utilisant des nœuds purement disjonctifs ou en « chaînant » les graphes.

Proposition 7.4.1 (Disjonction)**.** Soit $<$ un ordre strict total sur $\mathcal{I}$. SD, FSD et $\text{OSD}_<$ satisfont $\vee$**C** et $\vee$**BC**.

Proposition 7.4.2 (Conjonction)**.** SD et SDD satisfont $\wedge$**C** et $\wedge$**BC**.

Les termes et les clauses peuvent être représentés en temps polynomial dans n'importe lequel des langages de la famille SD.

Proposition 7.4.3 (Termes et clauses)**.** Soit L un des sous-langages de SD que nous avons défini ; on peut montrer que $\text{L} \leqslant_p \texttt{term}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}$ et $\text{L} \leqslant_p \texttt{clause}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}$.

De ces trois propositions, on tire des propriétés fondamentales sur la relation entre les langages de la famille SD, DNF et CNF.

Proposition 7.4.4. Soit $<$ un ordre strict total sur $\mathcal{I}$. On montre les relations suivantes :

$$\mathtt{OSD}_< \leqslant_p \mathtt{DNF}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}},$$
$$\mathtt{SDD} \leqslant_p \mathtt{CNF}_{\mathcal{I}}^{\mathbb{T}\mathbb{Z}}.$$

En particulier, on obtient que $\mathtt{OSD}_< \leqslant_p \mathtt{DNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$ et que $\mathtt{SDD} \leqslant_p \mathtt{CNF}_{\mathcal{B}}^{\mathbb{S}\mathbb{B}}$, ce qui a d'importantes conséquences sur la compacité, les requêtes et les transformations satisfaites.

Validité des FSDDs

Algorithme 7.2 Étant donné un FSDD φ, vérifie si φ est valide.

pour tout $x \in \mathrm{Scope}(\varphi)$ **faire**
  pour tout nœud N de φ **faire**
    soit $S_{x,N} \coloneqq \varnothing$
  soit $S_{x,\mathrm{Root}(\varphi)} \coloneqq \mathrm{Dom}(x)$
pour tout nœud N de φ, ordonnés de la racine au puits **faire**
  soit $x \coloneqq \mathrm{Var}(N)$
  si $\bigcup_{E \in \mathrm{Out}(N)} \mathrm{Lbl}(E) \not\supseteq S_{x,N}$ **alors**
    renvoyer faux
  pour tout $E \in \mathrm{Out}(N)$ **faire**
    soit $S_{x,\mathrm{Dest}(E)} \coloneqq S_{x,\mathrm{Dest}(E)} \cup S_{x,N} \cup \mathrm{Lbl}(E)$
    pour tout $y \in \mathrm{Scope}(\varphi) \setminus \{x\}$ **faire**
      soit $S_{y,\mathrm{Dest}(E)} \coloneqq S_{y,\mathrm{Dest}(E)} \cup S_{y,N}$
renvoyer vrai

Proposition 7.4.5 (Validité)**.** FSDD satisfait **VA**.

La preuve s'appuie sur l'algorithme 7.2, qui, pour schématiser, vérifie que « ce qui entre dans un nœud est égal à ce qui en sort ». Comme vérifier la validité d'une formule n'est pas polynomial sur DNF, cette propriété implique notamment que FSDD ne peut pas satisfaire $\vee$**C**.

Disjonction de FSDDs avec des clauses

Tout FSDD peut cependant être disjoint en temps polynomial avec toute clause.

Proposition 7.4.6 (Disjonction avec des clauses)**.** FSDD satisfait $\vee$**cl**.

Cette propriété, combinée à celle sur la validité, est la clef permettant de prouver que FSDD satisfait **IM**, ce qui a d'importantes conséquences sur la compacité, comme **CE**.

Négation de SDDs

Obtenir la négation d'un SDD peut se faire grâce à l'algorithme 7.3, qui « complémente » chaque nœud récursivement, en partant du puits.

Proposition 7.4.7 (Négation)**.** SDD, OSDD et $\texttt{OSDD}_<$ satisfont $\neg\mathbf{C}$.

Cependant, cette procédure ne maintient pas la convergence ; on ne sait en fait même pas si FSDD satisfait ou non $\neg\mathbf{C}$.

Algorithme 7.3 Construit la négation d'un SDD φ.

1 : **si** φ est vide **alors**
2 : **renvoyer** le graphe-puits
3 : **sinon**, **si** φ est le graphe-puits **alors**
4 : **renvoyer** le graphe vide
5 : soit ψ le graphe-puits
6 : soit $T \coloneqq \varnothing$ un ensemble de couples de nœuds, associant chaque nœud de φ à un nœud de ψ.
7 : **pour tout** nœud N de φ, ordonnés du puits à la racine, puits exclu **faire**
8 : créer un nœud N' étiqueté par la même variable x que N
9 : soit $U \coloneqq \mathrm{Dom}(x) \setminus \bigcup_{E \in \mathrm{Out}(N)} \mathrm{Lbl}(E)$
10 : **si** $U \neq \varnothing$ **alors**
11 : ajouter à N' un arc sortant E_{compl} étiqueté par U et pointant vers le puits de ψ
12 : **pour tout** $E \in \mathrm{Out}(N)$ **faire**
13 : soit $D = \mathrm{Dest}(E)$
14 : **si** il existe un couple $\langle D, D' \rangle \in T$ **alors** *// D a un correspondant D' dans ψ*
15 : ajouter à N' un arc sortant E' étiqueté par $\mathrm{Lbl}(E)$ et pointant vers D'
16 : **si** N' a au moins un arc sortant **alors**
17 : ajouter N' à ψ
18 : ajouter $\langle N, N' \rangle$ à T
19 : soit $R \coloneqq \mathrm{Root}(\varphi)$
20 : **si** il existe un couple $\langle R, R' \rangle \in T$ **alors** *// la racine a un correspondant*
21 : **renvoyer** ψ
22 : **sinon**
23 : **renvoyer** le graphe vide

7.4.2 Compacité

Théorème 7.4.8 (Compacité de la famille SD)**.** Les résultats du tableau 7.1 sont démontrés.

La figure 7.5 résume les résultats de compacité du théorème 7.4.8, ainsi que quelques autres résultats de compacité comparée entre les familles SD et BDD (voir

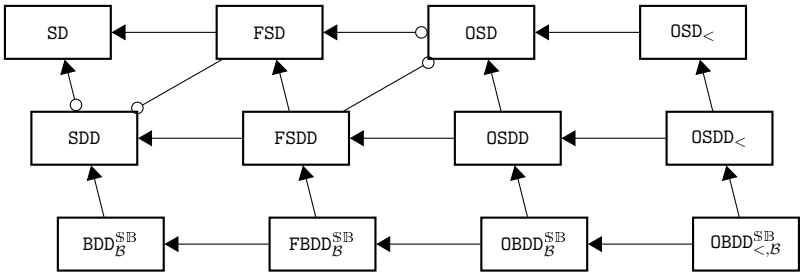


Fig. 7.5 : Compacité relative des familles SD et $\mathtt{BDD}_{\mathcal{B}}^{\mathbb{SB}}$. Sur un arc reliant deux langages $\mathtt{L}_1$ et $\mathtt{L}_2$, une flèche pointant vers $\mathtt{L}_1$, signifie que $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$; l'absence de symbole du côté de $\mathtt{L}_1$ (ni flèche ni cercle) signifie que $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$; et un cercle du côté de $\mathtt{L}_1$ signifie qu'on ne sait pas si $\mathtt{L}_1 \leqslant_s \mathtt{L}_2$ ou si $\mathtt{L}_1 \not\leqslant_s \mathtt{L}_2$ Les relations pouvant se déduire par transitivité ne sont pas représentées ; deux fragments non ancêtres l'un de l'autre sont donc incomparables du point de vue de la compacité.

L	SD	SDD	FSD	FSDD	OSD	OSDD	OSD$_<$	OSDD$_<$
SD	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
SDD	?	$\leqslant_s$	?	$\leqslant_s$	?	$\leqslant_s$	?	$\leqslant_s$
FSD	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
FSDD	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\not\leqslant_s^*$	$\leqslant_s$	$\not\leqslant_s^*$	$\leqslant_s$	$\not\leqslant_s^*$	$\leqslant_s$
OSD	$\not\leqslant_s$	$\not\leqslant_s$	?	?	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$	$\leqslant_s$
OSDD	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$
OSD$_<$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$	$\leqslant_s$
OSDD$_<$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\not\leqslant_s$	$\leqslant_s$

Tab. 7.1 : Résultats sur la compacité. Une étoile (*) indique un résultat dépendant du non-effondrement de la hiérarchie polynomiale PH.

la section 7.2.2). On peut y voir principalement qu'imposer la convergence, la décision exclusive ou l'ordonnancement peut mener à d'exponentielles augmentations de l'espace nécessaire. Expliquons rapidement les raisonnements permettant d'obtenir les résultats les plus cruciaux.

Les résultats dépendant de l'effondrement de la hiérarchie polynomiale proviennent tous du lemme 4.3.10, qui établit qu'aucune fonction de compilation en espace polynomial ne peut rendre polynomiale l'implication clausale sur les CNFs.

La preuve de $\mathtt{OSD}_< \not\leqslant_s \mathtt{OSDD}$ s'appuie sur la famille classique de fonctions $\bigwedge_{i=1}^{n} \big[y_i = z_i\big]$, dont la représentation peut être polynomiale pour certains ordres, mais est toujours exponentielle pour d'autres — ce qui prouve qu'au pire cas, on ne peut imposer un ordre donné sans perdre un espace exponentiel.

Pour prouver que $\mathtt{OSDD} \not\leqslant_s \mathtt{FSDD}$, on utilise la fonction représentant le problème de la n-coloration d'un graphe en étoile possédant n nœuds. La figure 7.6 est une illustration du cas $n = 3$: mettre la variable centrale à la racine donne un OSDD de taille polynomiale, tandis que la mettre en dernière position peut donner des

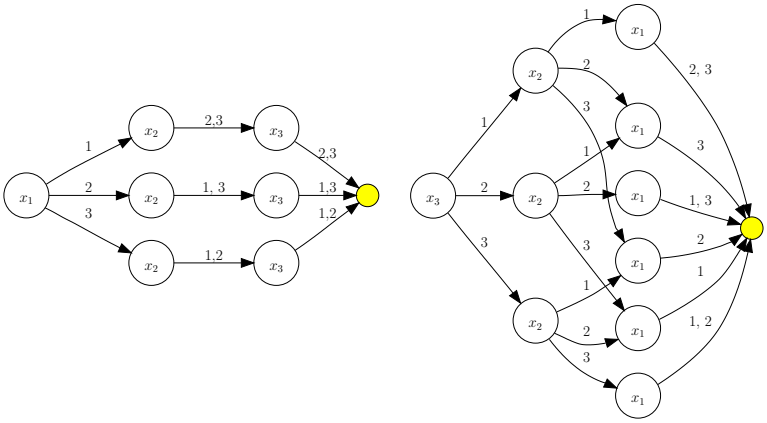


Fig. 7.6 : Le problème de la coloration d'un graphe en étoile, pour 3 variables et 3 couleurs. L'OSDD de droite est ordonné avec la variable centrale x_1 en dernière position, tandis que l'OSDD de gauche est ordonné avec x_1 en première position.

OSDDs de taille exponentielle. Nous utilisons ce fait pour construire un FSDD n'ayant aucun OSDD équivalent de taille polynomiale.

Enfin, $\texttt{OSDD} \not\leqslant_s \texttt{OSD}_<$ vient de la carte de compilation pour des variables booléennes [théorème 1.4.14] : si ce n'était pas vrai, on aurait $\texttt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}} \leqslant_s \texttt{DNF}^{\$\mathbb{B}}_{\mathcal{B}}$, ce qui est impossible.

7.4.3 Requêtes et transformations

Théorème 7.4.9. Les résultats des tableaux 7.2 et 7.3 sont démontrés.

Certains points sont à noter. Tout d'abord, comme c'est généralement le cas en compilation de connaissances, les langages satisfaisant beaucoup de requêtes satisfont peu de transformations, et *vice-versa*. Comme dans la carte de `IA`, il y a une similarité marquée entre les sous-langages convergents de `SD` et les sous-langages décomposables de `NNF` [théorème 1.4.15] : `FSD` et `DNNF` satisfont *le même ensemble de requêtes et transformations*, tout comme `FSDD` et `d-DNNF` — alors même que les SDs convergents ne sont *pas* des structures décomposables.

Nous pouvons aussi voir que `OSDD` et $\texttt{OSDD}_<$, étant des extensions directes de $\texttt{OBDD}^{\$\mathbb{B}}_{\mathcal{B}}$ et $\texttt{OBDD}^{\$\mathbb{B}}_{<,\mathcal{B}}$, ont sans surprise des propriétés très similaires. Il existe cependant un différence importante : les OBDDs booléens satisfont **SFO**, tandis que ce n'est pas le cas des OBDDs entiers. Cela vient du fait que la taille des domaines des variables est connue dans le cadre $\texttt{BDD}^{\$\mathbb{B}}_{\mathcal{B}}$: oublier une variable revient, en exploitant la décomposition de Shannon, à effectuer une disjonction binaire. Au contraire, dans le cas des SDs, les domaines ne sont pas bornés ; utiliser la même procédure nécessiterait une disjonction non bornée, qui n'est satisfaite ni par `OSDD` ni par $\texttt{OSDD}_<$.

L	CO	VA	MC	CE	IM	EQ	SE	MX	CX	CT	ME
SD	○	○	√	○	○	○	○	○	○	○	○
SDD	○	○	√	○	○	○	○	○	○	○	○
FSD	√	○	√	√	○	○	○	√	√	○	√
FSDD	√	√	√	√	√	?	○	√	√	?	√
OSD	√	○	√	√	○	○	○	√	√	○	√
OSDD	√	√	√	√	√	√	○	√	√	√	√
$\text{OSD}_<$	√	○	√	√	○	○	○	√	√	○	√
$\text{OSDD}_<$	√	√	√	√	√	√	√	√	√	√	√

Tab. 7.2 : Résultats sur les requêtes ; √ signifie « satisfait », et ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ ».

L	CD	TR	FO	SFO	EN	SEN	$\vee$C	$\vee$BC	$\vee$clC	$\wedge$C	$\wedge$BC	$\wedge$tC	$\neg$C
SD	√	○	○	√	○	√	√	√	√	√	√	√	?
SDD	√	○	○	√	○	√	√	√	√	√	√	√	√
FSD	√	√	√	√	○	○	√	√	√	○	○	√	○
FSDD	√	○	○	○	○	○	○	○	√	○	○	√	?
OSD	√	√	√	√	○	○	?	?	√	○	○	√	○
OSDD	√	•	•	•	•	•	•	○	√	•	○	√	√
$\text{OSD}_<$	√	√	√	√	○	○	√	√	√	○	√	√	○
$\text{OSDD}_<$	√	•	•	•	•	•	•	√	√	•	√	√	√

Tab. 7.3 : Résultats sur les transformations ; √ signifie « satisfait », • signifie « ne satisfait pas », et ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ ».

Notons également la ressemblance entre OSD et FSD. Ils satisfont exactement le même ensemble de requêtes et de transformations, excepté $\vee$**C** et $\vee$**BC**, pour lesquels le comportement de OSD est inconnu.

D'un point de vue plus pratique, si un utilisateur a besoin d'effectuer, sur sa forme compilée, de nombreuses requêtes mais aucune transformation à part du conditionnement, comme c'est le cas dans certaines applications de configuration, alors les FSDDs sont plus intéressants que les OSDDs. En effet, ils peuvent être beaucoup plus compacts, tout en satisfaisant presque les mêmes requêtes. Imposer un ordre sur les variables (c'est-à-dire passer de FSDD à OSDD) est avantageux dès qu'une des transformations $\neg$**C**, **SEN** ou **SFO** est nécessaire. En relâchant la contrainte de décision exclusive, on obtient le fragment FSD, qui satisfait en sus **FO** et $\vee$**C** ; FSD convient particulièrement aux applications telles que la planification, où il est nécessaire de régulièrement vérifier la cohérence, oublier des variables et extraire des modèles.

*
**

Après ces considérations théoriques sur les SDs, le chapitre 8 adoptera un point de vue plus pratique, en étudiant une méthode pour *compiler* des *set-labeled diagrams*.

CHAPITRE

8

Construction de *set-labeled diagrams*

Le chapitre 7 a identifié un certain nombre de sous-langages des *set-labeled diagrams* bien adaptés à notre application, à savoir la famille des FSDs. Nous avons vu que les FIAs pouvaient être transformés en FSDs en temps polynomial sans perte d'information. Dans ce chapitre, nous étudions comment il est possible de compiler des *set-labeled diagrams* directement à partir d'un réseau de contraintes sur des variables discrètes. Nous commençons par définir formellement ce langage d'entrée [§ 8.1], puis passons rapidement sur la compilation *bottom-up* [§ 8.2], et enfin présentons notre compilateur « CHOCO *with a trace* » [§ 8.3].

8.1 Réseaux de contraintes discrets

Comme nous l'avons vu dans la section 1.5.2, le réseau de contraintes [définition 1.5.1] est une manière naturelle et générique de représenter une base de connaissances. Le problème consistant à décider de la cohérence d'un réseau de contraintes sur des variables énumérées est généralement appelé *CSP discret* ; nous utilisons un nom similaire ici.

Définition 8.1.1. On appelle *réseau de contraintes discret* (DCN, *discrete constraint network*) un réseau de contraintes $\Pi = \langle V, \mathscr{C} \rangle$ tel que $V \subseteq \mathcal{E}$.

Notons que les variable d'un DCN ne sont pas strictement les mêmes que celles d'un SD : on ne considère ici que des intervalles *finis*. Cette distinction a deux avantages : elle reflète la définition la plus répandue d'un CSP discret, et surtout, elle permet de compiler n'importe quel DCN vers SD, puisque $\mathtt{SD}_{\mathcal{E}}$ est complet [proposition 7.2.6]. Ainsi, compiler l'ensemble solution d'un CN discret en SD ne

nécessite aucune approximation, contrairement au cas de la compilation d'un CN continu en IA [§ 5.1].

Les contraintes sont simplement définies comme des ensembles d'instanciations ; elles ne sont cependant pas systématiquement *données* sous cette forme.

- Une contrainte peut être donnée *en extension*, c'est-à-dire par une liste de tuples de valeurs *admissibles*. Cela correspond directement à la définition.
- Une contrainte peut être donnée *en intension*, c'est-à-dire par une *formule* sur ses variables : les tuples admissibles sont alors ceux qui sont cohérents avec la formule.

Les contraintes données en extension peuvent être vues comme des formules en DNF — et en tant que telles, peuvent être transformées en SDs en temps polynomial [proposition 7.4.4]. En revanche, les contraintes données en intension ne sont pas traduisibles en SDs de manière directe : c'est le cas par exemple des contraintes $x < y$, $x \times y = z$ ou alldiff$(x_1, \ldots, x_n)$.

Pour compiler un DCN vers `SD`, nous choisissons de transformer ce type de contraintes en formules DNF. Dans ce but, nous utilisons un *solveur de CNs discrets* en tant qu'outil d'aide à la construction de SDs. Contrairement au solveur de CNs continus RealPaver que nous avons utilisé pour construire des automates à intervalles, les solveurs de CNs discrets renvoient l'ensemble *exact* des solutions du CN qui leur est donné en entrée. Il n'y a donc aucun problème lié à l'approximation dans ce chapitre. Le solveur que nous avons utilisé en pratique est CHOCO [CHO10], un solveur *open-source* écrit en Java qui s'appuie sur de nombreuses techniques de satisfaction de contraintes identifiées dans la littérature.

8.2 Compilation *bottom-up*

Mis à part le fait que l'étape de résolution fournit l'ensemble exact des solutions plutôt qu'une approximation, l'utilisation de la sortie de CHOCO pour compiler des SDs ressemble beaucoup à l'utilisation de la sortie de RealPaver pour compiler des IAs [§ 5.2]. CHOCO renvoie l'ensemble *énuméré* des solutions du DCN d'entrée, c'est-à-dire qu'il renvoie une liste de toutes les solutions. Il n'y a pas de boîte dans ce cadre, seulement des instanciations — qui sont elles-même des cas particuliers de boîtes, et par conséquent polynomialement traduisibles vers n'importe quel sous-langage de `SD`, comme montré dans la proposition 7.4.3. Cependant, pour obtenir la même fonction booléenne que celle représentée par le SD de départ, il est nécessaire d'effectuer la disjonction de tous les SDs-instanciations. Lors de la compilation de SDs ordonnés, il faut veiller à ce que l'ordre des variables de tous ces SDs soit le même. Il est à noter que les OSDDs compilés en utilisant cette méthode auront une taille probablement bien supérieure à celle des OSDs obtenus de manière similaire, étant donné qu'`OSDD` ne supporte que la disjonction *bornée*.

Se baser sur la sortie de CHOCO est la manière la plus simple de compiler un DCN en SD. Il est également possible, d'après Amilhastre [Ami99], de compiler un SD pour chaque contrainte dans le DCN, puis de combiner ces SDs élémentaires en un SD plus gros représentant la conjonction de toutes les contraintes, c'est-à-dire le DCN tout entier. Cette technique n'était pas utilisable sur `FIA`, qui ne supporte pas la conjonction, même bornée ; mais à condition que toutes les contraintes soient compilées avec le même ordre des variables, elle est applicable à $\texttt{OSD}_<$ et $\texttt{OSDD}_<$.

Comme expliqué dans la section 1.5.2.1, l'inconvénient de ce type de méthode est qu'elle génère des structures de données intermédiaires, qui consomme de l'espace et peuvent même être de taille exponentiellement plus grande que le diagramme de décision final. C'est pourquoi nous nous penchons, dans la section suivante, sur une méthode alternative basée sur la *trace* de CHOCO, comme nous le fîmes avec RealPaver [§ 5.3].

8.3 CHOCO *with a Trace*

Appliquons donc les principes de « DPLL *with a trace* » [HD05] au solveur CHOCO. L'idée d'utiliser la trace d'un solveur a été adaptée par Hadzic et al. [HH$^+$08] à la compilation approchée de MDDs. Une technique similaire est utilisée par Mateescu, Dechter et Marinescu [MDM08], où des AOMDDs (c'est-à-dire des MDDs avec des nœuds « et ») sont construits en suivant l'arbre d'une recherche « et/ou ».

Ces approches utilisent un ordre de variables prédéterminé (ordre arborescent dans le cas des AOMDDs), et les variables ne peuvent être répétées le long d'un chemin. Nous relâchons ces hypothèses ici : le choix de la variable suivante sur laquelle brancher peut être fait *dynamiquement*, en fonction d'une heuristique, et les domaines ne sont pas nécessairement divisés en singletons — ce qui implique qu'une même variable peut être rencontrée plusieurs fois le long d'un chemin dans l'arbre de recherche. En effet, il n'est pas toujours nécessaire de compiler des SDs ordonnés ; la convergence suffit pour les applications ne nécessitant que du conditionnement et de l'extraction de modèle, comme c'est le cas de l'exploitation d'une politique de décision.

8.3.1 Algorithme de recherche de CHOCO

L'algorithme 8.1 est une version simplifiée de la procédure de recherche de CHOCO. Il s'agit d'une recherche classique en profondeur d'abord, avec de la propagation de contraintes à chaque nœud. Nous la présentons de manière récursive pour clarifier son fonctionnement.

Elle prend en entrée un réseau de contraintes $\Pi = \langle X, \mathcal{C} \rangle$, défini par un ensemble de contraintes $\mathcal{C}$ sur un ensemble de variables X [définition 1.5.1], ainsi qu'un ensemble de variables assignées (qui est évidemment vide lors de l'appel

initial de la procédure). Elle renvoie l'ensemble énuméré des solutions de Π, c'est-à-dire que chaque solution est explicitement retourné.

Algorithme 8.1 `CHOCO`(Π, X_a) : renvoie l'ensemble de toutes les solutions du CN discret Π. X_a est l'ensemble courant des variable assignées.

```
1 : Prune(Π)
2 : si Π est prouvé incohérent alors
3 :     renvoyer ∅
4 : si X_a = X alors   // toutes les variables de Π sont assignées
5 :     soit x⃗ l'instanciation correspondante
6 :     renvoyer {x⃗}
7 : x := Sel_var(Π)
8 : R := Split(Π, x)
9 : S := ∅
10 : pour tout r ∈ R faire
11 :     soit X'_a := X_a
12 :     si r est un singleton alors   // x est maintenant assignée
13 :         X'_a := X'_a ∪ {x}
14 :     S_r := CHOCO(Π_|x∈r, X'_a)
15 :     S := S ∪ S_r
16 : renvoyer S
```

L'algorithme 8.1 commence par faire appel à la fonction interne `Prune` sur le DCN courant ; c'est l'étape de *propagation*, où les valeurs dont il est prouvé qu'elles sont incompatibles avec certaines contraintes sont retirées des domaines des variables. La fonction `Prune` fait usage de diverses techniques de propagation de contraintes, que nous ne détaillons pas ici [voir guide utilisateur de CHOCO : CHO10]. On récupère un DCN modifié (ligne 1), qui est garanti d'avoir le même ensemble solution que l'entrée Π.

Ensuite, la procédure vérifie si elle a atteint une feuille de l'arbre de recherche : c'est le cas (i) si le DCN courant est incohérent (c'est-à-dire qu'au moins une variable a un domaine vide), auquel cas l'ensemble vide est renvoyé, ou (ii) si toutes les variables ont été assignées, auquel cas l'instanciation correspondante est une solution.

Le reste de la procédure s'occupe des nouveaux nœuds de l'arbre de recherche. Une variable x est choisie grâce à la fonction `Sel_var` ; le choix dépend d'une heuristique sélectionnée par l'utilisateur, mais seules les variables non-assignées peuvent se voir choisies. Le domaine courant de la variable choisie est alors partitionné par la fonction `Split`. L'utilisateur peut contrôler en amont le nombre d'éléments de la partition et la taille de chaque élément. Ensuite, pour chacun des éléments r de la partition, la procédure s'appelle récursivement sur le DCN Π dans lequel $\mathrm{Dom}(x)$ a été restreint à r. Si r est un singleton, la variable x est marquée comme étant assignée ; elle ne sera plus jamais modifiée dans ce sous-arbre de recherche.

Algorithme 8.2 `CHOCO_to_SD`(Π, X_a) : renvoie un SD représentant l'ensemble solution du CN discret Π. X_a est l'ensemble courant des variables assignées.

```
 1 : Prune(Π)
 2 : si Π est prouvé incohérent alors
 3 :     renvoyer le SD vide
 4 : si X_a = X alors   // toutes les variables de Π sont assignées
 5 :     renvoyer le SD puits
 6 : x ≔ Sel_var(Π)
 7 : R ≔ Split(Π, x)
 8 : Ψ ≔ ∅
 9 : pour tout r ∈ R faire
10 :     soit X'_a ≔ X_a
11 :     si r est un singleton alors   // x est maintenant assignée
12 :         X'_a ≔ X'_a ∪ {x}
13 :     soit ψ_r ≔ CHOCO_to_SD(Π_|x∈r, X'_a)
14 :     ajouter à Ψ le couple ⟨r, ψ_r⟩
15 : soit le nœud N ≔ Get_node(x, Ψ)
16 : soit ψ le graphe de racine N
17 : renvoyer ψ
```

La procédure ainsi décrite n'est pas spécifique à CHOCO. Notons en particulier que l'implémentation réelle des fonctions internes `Prune`, `Split` et `Sel_var` importe peu, du moment qu'elles remplissent les conditions.

8.3.2 Tracer CHOCO

Notre approche consiste à suivre le déroulement de l'algorithme 8.1 en construisant des *set-labeled diagrams* au lieu de simplement énumérer l'ensemble solution. Comme nous l'avons fait dans la section 5.3.2, nous présentons la procédure modifiée (algorithme 8.2) en encadrant les différences.

Le SD construit par l'algorithme 8.2 reflète fidèlement l'arbre de recherche. Il n'a pas besoin de construire des « nœuds d'élagage » comme l'algorithme 5.3 [p. 107], car il n'y a pas de paramètre gérant la précision : la recherche se poursuit toujours jusqu'à ce que chaque variable soit explicitement assignée.

Quand la procédure atteint une feuille de l'arbre de recherche, il s'agit soit d'une solution, soit d'un cul-de-sac ; dans le premier cas, l'algorithme renvoie le SD puits (ligne 5), pour rendre compte du fait que toute instanciation est solution du problème courant ; et dans le deuxième cas, elle renvoie le SD vide (ligne 3).

Dans le cas d'un nœud de recherche interne, une variable non-assignée x est choisie, et son domaine courant partagé. L'exploration de l'arbre de recherche associé à chaque élément r de la partition renvoie un SD ψ_r, qui représente l'ensemble solution du sous-problème $\Pi_{|x \in r}$ obtenu en réduisant le domaine de x à r.

L'ensemble solution du réseau de contraintes courant Π est alors un SD ψ, ayant pour racine un nœud x avec pour chaque r un arc sortant étiqueté par r et pointant vers ψ_r. Ce nœud est obtenu ligne 15 grâce à une fonction interne `Get_node`, qui fonctionne de la même façon que la fonction `Get_node` de la section 5.3.2. En particulier, elle n'ajoute pas d'arc r si le SD pointé ψ_r est vide, et elle se charge d'effectuer le maximum possible d'opérations de réduction à la volée. Le SD compilé est ainsi notamment garanti de ne contenir aucun couple de nœuds isomorphes, ceci en raison de la *table de nœuds uniques* que maintient `Get_node`.

Comme pour l'algorithme 5.2 [p. 105], à tout moment durant la compilation, le SD courant est toujours de taille inférieure au SD final (mais il faut noter que ce SD final n'est généralement pas réduit, car `Get_node` ne peut pas traiter les nœuds bégayants). La procédure est par conséquent en espace polynomial selon la taille de sa sortie — ce qui n'est pas le cas pour l'approche *bottom-up*.

8.3.3 Cache des sous-problèmes

En nous inspirant à nouveau de Huang et Darwiche [HD05], nous essayons de réduire la complexité temporelle du compilateur, de manière à ce qu'elle soit plus proportionnée à la taille de la forme compilée, en utilisant un *cache*.

Principe et algorithme

L'objectif est d'éviter d'avoir à explorer des *sous-problèmes équivalents*. Les modifications sont indiquées dans l'algorithme 8.3 ; concrètement, à chaque fois qu'un sous-probème Π est résolu, il est enregistré dans une table de hachage à une clef k, qui dépend des domaines courants des variables (ligne 20). De plus, avant de traiter un sous-problème Π', le compilateur calcule sa clef k' (ligne 2), vérifie si elle est déjà présente dans la table de hachage et, le cas échéant, renvoie le SD correspondant (ligne 4).

L'idée est que puisque les sous-problèmes en question sont équivalents, l'exploration du sous-arbre de recherche pour le second sous-problème est inutile : le SD obtenu serait de toute façon équivalent à celui déjà construit (et enregistré dans le cache). L'utilisation d'un cache transforme en quelque sorte l'arbre de recherche en *graphe* de recherche. Il est à noter que le cache concerne des *problèmes*, et n'a de ce fait rien à voir avec la table de nœuds uniques, dont le but est d'éviter la création de nœuds isomorphes.

Calcul de la clef de hachage

Soit Π le réseau de contraintes courant dans un nœud de recherche quelconque, et X_a l'ensemble des variables déjà assignées dans Π. L'exploration du sous-arbre de recherche ayant pour racine le nœud courant renvoie un SD, qui ne porte que sur les variables de $X \setminus X_a$, et représente l'ensemble solution de Π restreint à $X \setminus X_a$. Le réseau de contraintes courant Π' pour un nœud de recherche différent est équivalent à Π si et seulement si les SDs correspondants sont équivalents — c'est-à-dire, si et seulement si leurs ensembles de solutions, projetés sur $X \setminus X_a$, sont égaux.

Algorithme 8.3 `SD_builder`(Π, X_a) : renvoie un SD représentant l'ensemble solution du CN discret Π. X_a est l'ensemble courant des variables assignées.

```
1 : Prune(Π)
2 : k ≔ Compute_key(Π, X_a)
3 : si le cache contient une entrée pour la clef k alors
4 :       renvoyer le SD correspondant à la clef k dans le cache
5 : si Π est prouvé incohérent alors
6 :       renvoyer le SD vide
7 : si X_a = X alors   // toutes les variables de Π sont assignées
8 :       renvoyer le SD puits
9 : x ≔ Sel_var(Π)
10 : R ≔ Split(Π, x)
11 : Ψ ≔ ∅
12 : pour tout r ∈ R faire
13 :       soit X'_a ≔ X_a
14 :       si r est un singleton alors   // x est maintenant assignée
15 :             X'_a ≔ X'_a ∪ {x}
16 :       soit ψ_r ≔ CHOCO_to_SD(Π_|x∈r, X'_a)
17 :       ajouter à Ψ le couple ⟨r, ψ_r⟩
18 : soit le nœud N ≔ Get_node(x, Ψ)
19 : soit ψ le graphe de racine N
20 : enregistrer ψ à la clef k dans le cache
21 : renvoyer ψ
```

Comme nous cherchons à utiliser le cache pour éviter d'explorer plusieurs fois des sous-problèmes équivalents, les clefs de hachage doivent être étudiées pour grouper autant de sous-problèmes équivalents que possible. Idéalement, *tous* les sous-problèmes équivalents sont associés à une même clef — l'algorithme n'ayant ainsi *jamais* à construire un graphe équivalent à l'un de ceux qu'il a déjà retournés. Cependant, pour que le cache soit utilisable efficacement, la clef doit être aussi courte que possible et facile à calculer ; un compromis est donc nécessaire.

Intuitivement, le sous-problème courant peut être représenté par une clef listant le domaine courant de chacune des variables. Il est possible de réduire la taille de cette clef, en utilisant la même technique que Lecoutre et al. [LS$^+$07]. L'idée est de retirer certaines variables de cette liste ; une variable x est retirée si (i) elle est assignée ($x \in X_a$), et (ii) elle n'intervient que dans des contraintes *nécessairement satisfaites* dans le sous-problèmes courant (de telles contraintes sont parfois dites *universelles* ou *impliquées*). Lecoutre et al. [LS$^+$07] ont prouvé que ce mécanisme préservait l'ensemble des solutions.

Avantages du cache

L'utilisation d'un cache est intéressante pour les raisons décrites par Huang et Darwiche [HD05] — cela permet de rendre la complexité temporelle du compila-

teur polynomiale en la taille de la sortie. Cet avantage n'est pas spécifiquement lié à l'aspect « compilation » de la procédure : par exemple, Lecoutre et al. [LS+07] appliquent cette même idée à des procédures de résolution de CSPs, dont l'objectif est de ne trouver qu'une seule solution. Ils l'utilisent en revanche de manière un peu différente, puisque comme leur algorithme s'arrête dès qu'une solution est trouvée, ils n'enregistrent que des sous-problèmes incohérents dans leur cache.

Un autre avantage du cache, lié cette fois à la compilation, est que si le sous-problème courant est cohérent, stocker son SD peut permettre de réduire la taille du SD final. En effet, sans cache, les sous-problèmes équivalents sont explorés indépendamment, avec des ordres de variables possiblement différents, en particulier si les heuristiques de sélection de variable et de partitionnement de domaine se basent sur du hasard. Cela peut donc conduire à avoir, dans la forme compilée, plusieurs sous-graphes représentant la même fonction booléenne mais n'étant pas isomorphes. Ces sous-graphes ne pouvant être fusionnés par réduction, le SD final est plus grand que celui obtenu en utilisant le cache.

Minimisation du cache

Devoir garder trace de chaque sous-problème peut conduire à un cache gigantesque. Pour minimiser l'espace mémoire, nous avons choisi de limiter la taille du cache à une valeur arbitraire. Plus précisément, à chaque fois que le cache est plein, un certain nombre d'entrées sont retirées, le but étant de ne garder que les plus intéressantes. Nous avons utilisé l'heuristique suivante : on rejette d'abord les entrées qui ont été le moins utilisées, puis les plus anciennes, et enfin celles ayant les plus longues clefs (qui correspondent au sous-problèmes les plus petits).

Ces techniques de minimisation du cache réduisent la quantité d'espace mémoire nécessaire au compilateur, et permettent d'accélérer les opérations liées au cache. Elles permettent donc de compiler des problèmes plus gros, mais n'améliorent pas les SDs résultants — c'est même l'inverse, puisque comme nous l'avons montré dans la sous-section précédente, l'utilisation du cache peut réduire la taille de la forme compilée.

8.3.4 Propriétés des SDs compilés

Structure

Pour les mêmes raisons que « RealPaver *with a trace* » [voir § 5.3.5.1], les *set-labeled diagrams* renvoyés par l'algorithme 8.3 sont toujours convergents. En effet, les domaines des variables ne peuvent que rétrécir, que ce soit par branchement ou par propagation. Cependant, contrairement aux FIAs obtenus en section 5, les SDs compilés satisfont toujours la décision exclusive : la fonction `Split` construit en effet une *partition* du domaine courant de la variable. Comme les éléments d'une partition sont forcément *disjoints*, l'algorithme 8.3 renvoie toujours des FSDDs.

Néanmoins, en fonction du comportement des fonctions `Sel_var` et `Split`, la structure de ces FSDDs peut s'avérer plus spécifique.

- Si `Split` branche sur des *singletons*, chaque variable est forcément assignée la première fois qu'elle est choisie ; les SDs résultants satisfont donc la propriété de *read-once*.
- Si, en plus, `Sel_var` suit un ordre statique $<$ sur les variables, le résultat est une $\texttt{OSDD}_{<}$-représentation.

Variables

Il est à noter que notre compilateur, étant basé sur CHOCO, est intrinsèquement limité au domaine d'interprétation des entrées autorisées par CHOCO. Par conséquent, il ne peut manipuler que des variables *énumérées*, alors que les SDs peuvent potentiellement porter sur des variables à domaine non-borné. « CHOCO *with a trace* » ne peut donc compiler que des $\texttt{SD}_{\mathcal{E}}$-représentations. Puisque $\texttt{MDD} = \texttt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$ [voir § 7.2.2], on pourrait s'attendre à ce qu'en branchant sur des singletons et en suivant un ordre de variables statique, « CHOCO *with a trace* » renvoie des MDDs ; c'est sans compter avec l'opération de réduction des arcs contigus appliquée dans la fonction `Get_node`, qui peut fusionner des arcs, changeant donc l'expressivité littérale des structures de $\$\mathbb{Z}$ à $\mathbb{T}\mathbb{Z}$. Ainsi paramétré, « CHOCO *with a trace* » fournit donc en réalité des $\texttt{OSDD}_{\mathcal{E}}$-représentations, mais pas des $\texttt{OSDD}_{\mathcal{E}}^{\$\mathbb{Z}}$-représentations (i.e. des MDDs).

*
**

Nous avons implanté « CHOCO *with a trace* » en ajoutant au code source de CHOCO les éléments présentés dans ce chapitre. Notre compilateur nous a permis d'obtenir des résultats expérimentaux pour nos applications ; c'est l'objet du chapitre suivant.

CHAPITRE

9

Expérimentations sur les *set-labeled diagrams*

Il est possible d'utiliser des *set-labeled diagrams* pour discrétiser des automates à intervalles. Nous avons montré dans le chapitre 7 que le langage des SDs convergents et ses sous-langages supportent les requêtes et transformations les plus importantes pour les applications de planification basées sur la compilation. En particulier, cela implique que discrétiser des IAs ne pose aucun problème, puisque les SDs sont aussi efficaces que les IAs en ce qui concerne leur manipulation en ligne. Cette efficacité est également intéressante pour les problèmes ne portant que sur des variables énumérées, comme nos *benchmarks ObsToMem* et *Telecom* ; c'est pourquoi nous avons étudié, dans le chapitre 8, des méthodes de compilation de réseaux de contraintes discrets en SDs.

Dans le présent chapitre, nous présentons nos manipulations expérimentales des *set-labeled diagrams* : nous donnons quelques détails sur la bibliothèque que nous avons développée [§ 9.1], puis montrons nos résultats expérimentaux de compilation [§ 9.2], et enfin nos résultats quant à l'utilisation pratique des SDs [§ 9.3].

9.1 Implémentation

9.1.1 Cadre expérimental

Nous avons écrit en Java une bibliothèque permettant à un utilisateur de manipuler des *set-labeled diagrams*. Elle fonctionne d'une manière similaire à notre bibliothèque sur les IAs [§ 6.1], s'appuyant notamment sur une fonction `Get_node` pour construire des SDs incrémentalement tout en appliquant des opérations de réduction à la volée. Notre bibliothèque est capable de traiter les huit sous-langages de `SD` que nous avons définis au chapitre 7. Cela est rendu possible par un système

de *propriétés de graphes* : le programme est par exemple capable de reconnaître qu'un SD est convergent — et de choisir alors une implémentation des requêtes et transformations adaptée à cette spécificité. Ainsi, supposons que l'utilisateur ait besoin d'oublier des variables sur un SD : la méthode de base pour faire ceci est de construire une disjonction de graphes conditionnés, cependant si le SD est convergent, il existe une méthode bien plus efficace, que notre programme peut appliquer automatiquement.

9.1.2 Compilateur de *set-labeled diagrams*

Il y a trois façons de construire un SD en utilisant notre paquetage : « à la main », avec la fonction `Get_node` ; en discrétisant un automate à intervalles ; et en utilisant « CHOCO *with a trace* ». La procédure de discrétisation fonctionne de la manière décrite dans la section 7.3.2, c'est-à-dire en calculant des maillages pour l'IA considéré et en associant des valeurs discrètes à chaque élément de chaque maillage.

Nous avons modifié le code du solveur de CSPs CHOCO pour qu'il construise un FSDD parallèlement à sa recherche de solution, comme nous l'avons détaillé dans la section 8.3. Nous avons en particulier implanté le cache des sous-problèmes, qui a une influence sur la procédure de recherche de CHOCO en ce qu'il l'empêche d'explorer plusieurs fois des sous-problèmes équivalents. Nous avons également implanté des heuristiques guidant le solveur dans son choix de la prochaine variable sur laquelle brancher, notamment les heuristiques décrites par Amilhastre [Ami99], à savoir **HBW**, **HSBW** et **MCSInv**, qui choisissent la variable suivante en fonction de celles déjà choisies et des variables auxquelles elles sont liées. Ces trois heuristiques sont *statiques*, c'est-à-dire que l'ordre des variables est calculé une fois pour toutes avant le début de la compilation. Nous les avons modifiées pour les rendre *dynamiques* : elles choisissent alors la variable suivante en fonction de l'état *courant* du graphe de contraintes.

9.1.3 Opérations sur les *set-labeled diagrams*

Nous avons implanté un certain nombre d'opérations sur les SDs — principalement les opérations polynomiales, mais pas seulement. Les requêtes disponibles sont les suivantes : cohérence (**CO**) sur `FSD`, vérification de modèle (**MC**) sur tous les langages, équivalence (**EQ**) sur tous les langages, extraction de modèle (**MX**) sur tous les langages, extraction de contexte (**CX**) sur `FSD`, comptage de modèles (**CT**) sur tous les langages, et énumération de modèles (**ME**) sur tous les langages. Parmi les transformations disponibles, on trouve le conditionnement (**CD**) sur tous les langages, l'oubli (**FO**) sur `FSD` et `OSD`, la disjonction ($\vee$**C**) sur `SD`, `FSD` et $\texttt{OSD}_<$, la conjonction ($\wedge$**C**) sur `SD` et `SDD`, la conjonction binaire ($\wedge$**BC**) sur $\texttt{OSD}_<$ et $\texttt{OSDD}_<$, et la conjonction avec un terme ($\wedge$**tC**) sur tous les langages.

Nous avons de surcroît implanté les requêtes « spéciales » définies en section 6.1.3, à savoir **CDCO** (cohérence après conditionnement), **CDMX** (extraction de

modèle après conditionnement), et **CDFOMX** (extraction de modèle après conditionnement et projection). Une quatrième requête spéciale s'ajoute à celles-ci, dont le but est d'extraire le contexte d'un ensemble de variable donné dans un SD restreint à un terme donné. Nous appelons cette requête **TRCX**. Elle peut s'avérer utile par exemple sur une politique de décision, pour vérifier les valeurs des variables d'action qui restent permises après une observation peu fiable. Elle peut également être utile dans d'autres domaines, notamment la configuration, où extraire des contextes est particulièrement important.

9.2 Tests de compilation

problème	durée (ms)	#nœuds	#arcs	taille carac.	fichier (octets)
*Drone*4-30-3	3397	217	315	337	10 901
*Drone*7-30-3	11 146	625	929	953	28 957
*Drone*10-30-3	69 092	2145	3145	3239	97 941
*Drone*13-30-3	1 574 836	10 222	14 787	14 824	472 074
*ObsToMem*1-2-4-1-2	1796	308	383	416	13 895
*ObsToMem*1-2-4-2-4	2246	279	358	389	12 954
*ObsToMem*2-3-6-2-3	15 843	9515	11 415	11 893	385 461
*ObsToMem*2-3-6-3-6	23 002	4580	5612	5817	188 010
*ObsToMem*3-4-8-1-2	63 790	92 778	104 505	106 389	3 836 096
*ObsToMem*3-4-8-1-3	180 379	125 361	144 563	148 927	5 303 145
*ObsToMem*3-4-8-1-4	438 697	126 015	147 562	153 035	5 377 446
*ObsToMem*3-4-8-1-6	504 718	60 078	73 664	76 173	2 587 457
*ObsToMem*3-4-8-2-4	555 211	128 317	152 098	158 695	5 517 311
*Telecom*3-5-5-32	19 047	1631	2234	2256	68 021
*Telecom*3-5-6-40	34 087	2721	3865	3888	115 796
*Telecom*3-5-6-64	107 933	4661	7390	7413	209 898
*Telecom*4-5-4-33	37 207	2115	3010	3035	89 883
*Telecom*4-5-5-43	119 785	3906	5855	5881	170 737
*Telecom*4-6-5-56	715 574	8405	13 671	13 698	384 139
*Telecom*4-6-6-63	1 324 677	13 763	21 592	21 620	624 958

Tab. 9.1 : Résultats obtenus avec « CHOCO *with a trace* ».

Nous avons compilé avec « CHOCO *with a trace* » des instances des *benchmarks Drone* (dans sa version discrète), *ObsToMem* et *Telecom*, sur un ordinateur portable standard. Nous avons utilisé l'heuristique dynamique **HBW** et une fonction de branchement qui énumère le domaine courant ; en pratique, nous avons donc compilé des SDDs *read-once* non-ordonnés. Les résultats sont présentés dans le tableau 9.1 ; ils incluent la durée de compilation, le nombre de nœuds et d'arcs, la

problème	**CD** $\vec{s}$	**FO** $\mathcal{S}'$	**MX** $\hookrightarrow\vec{a}$	**CD** $\vec{a}$	**MX** $\hookrightarrow\vec{s}'$	**CDFOMX** $\langle\vec{s},\mathcal{S}'\rangle\hookrightarrow\vec{a}$	**CDMX** $\vec{a}\hookrightarrow\vec{s}'$
*Drone*4-3-30	28	16	0	14	1	1	1
*Drone*7-3-30	99	39	2	21	1	1	2
*Drone*10-3-30	72	11	0	9	0	4	5
*Drone*13-3-30	553	34	1	22	1	74	59
*OTM*1-2-4-1-2	37	10	1	4	0	1	1
*OTM*1-2-4-2-4	110	60	13	24	0	1	1
*OTM*2-3-6-2-3	429	26	2	6	0	12	4
*OTM*2-3-6-3-6	270	78	17	26	0	33	2
*OTM*3-4-8-1-2	5619	71	39	26	0	235	11
*OTM*3-4-8-1-3	10 516	463	74	171	0	127	19
*OTM*3-4-8-1-4	5148	1258	63	156	0	30	2
*OTM*3-4-8-1-6	3160	509	96	1329	0	25	4
*OTM*3-4-8-2-4	6891	68	13	17	0	32	3

Tab. 9.2 : Résultats pour le scénario 1 sur plusieurs instances des relations de transition de *Drone* et *ObsToMem*, obtenues avec « CHOCO *with a trace* ». Toutes les durées sont en millisecondes.

taille caractéristique, et la taille d'un fichier contenant une version texte du SD. Toutes les instances de *Drone* ont un temps alloué fixé à 30 unités, et 3 billes ; le paramètre qui varie est le nombre de zones. Les instances de *ObsToMem* sont indicées par le nombre de lignes de détection, le nombre de COMs et le nombre de banques mémoire ; les deux derniers paramètres sont le nombre maximal considéré de pannes de COMs et de banques mémoire, respectivement. Les instances de *Telecom* sont indicées par le nombre de canaux d'entrée, d'amplificateurs, de canaux de sortie, et de chemins disponibles.

Les propriétés des formes compilées sont satisfaisantes ; elles sont relativement compactes et, les plus larges instances exceptées, pourraient sans doute être embarquées dans des systèmes autonomes. Cependant, la compilation est plus longue que ce à quoi on pourrait s'attendre au vu de la taille des formes compilées. Dans le chapitre 6, avec « RealPaver *with a trace* », nous avons compilé des graphes plus gros en un temps plus court ; ceci pourrait venir de notre implémentation du cache des sous-problèmes. Néanmoins, ces résultats montrent que cette approche n'est pas sans intérêt.

9.3 Tests applicatifs

Dans cette section, nous donnons des résultats concernant l'utilisation en ligne des formes compilées. De la même façon que dans la section 6.3, nous simulons des séquences réalistes d'opérations dépendant de la nature du problème compilé.

problème	**CD** $\vec{s}$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}'$	**CDFOMX** $\langle \vec{s}, \mathcal{A} \rangle \hookrightarrow \vec{s}'$
*Drone*4-3-30	28	11	1	0
*Drone*7-3-30	99	40	4	1
*Drone*10-3-30	72	14	1	5
*Drone*13-3-30	553	50	5	67
*ObsToMem*1-2-4-1-2	37	6	0	1
*ObsToMem*1-2-4-2-4	110	54	1	1
*ObsToMem*2-3-6-2-3	429	27	2	22
*ObsToMem*2-3-6-3-6	270	54	1	10
*ObsToMem*3-4-8-1-2	5619	146	1	117
*ObsToMem*3-4-8-1-3	10 516	336	1	134
*ObsToMem*3-4-8-1-4	5148	285	1	38
*ObsToMem*3-4-8-1-6	3160	345	1	19
*ObsToMem*3-4-8-2-4	6891	43	1	32

Tab. 9.3 : Résultats pour le scénario 2 sur plusieurs instances des relations de transition de *Drone* et *ObsToMem*, obtenues avec « CHOCO *with a trace* ». Toutes les durées sont en millisecondes.

problème	**CD** $\vec{s}'$	**FO** $\mathcal{A}$	**MX** $\hookrightarrow \vec{s}$	**CDFOMX** $\langle \vec{s}', \mathcal{A} \rangle \hookrightarrow \vec{s}$	**CD** $\vec{s}'$	**MX** $\hookrightarrow \vec{s} \,.\, \vec{a}$	**CDMX** $\vec{s}' \hookrightarrow \vec{s} \,.\, \vec{a}$
*Drone*4-3-30	24	6	1	4	24	1	1
*Drone*7-3-30	90	27	4	2	90	7	1
*Drone*10-3-30	122	38	3	1	122	7	1
*Drone*13-3-30	1001	476	57	7	1001	64	4
*OTM*1-2-4-1-2	43	22	2	0	43	3	0
*OTM*1-2-4-2-4	137	100	5	1	137	17	3
*OTM*2-3-6-2-3	773	378	29	1	773	87	1
*OTM*2-3-6-3-6	394	196	15	1	394	54	1
*OTM*3-4-8-1-2	8223	2728	175	1	8223	877	2
*OTM*3-4-8-1-3	15 413	10 455	512	2	15 413	3577	3
*OTM*3-4-8-1-4	9908	6782	319	0	9908	1791	1
*OTM*3-4-8-1-6	6567	4376	261	2	6567	1346	1
*OTM*3-4-8-2-4	9739	3263	309	0	9739	1144	1

Tab. 9.4 : Résultats pour les scénarios 3 et 4 sur plusieurs instances des relations de transition de *Drone* et *ObsToMem*, obtenues avec « CHOCO *with a trace* ». Toutes les durées sont en millisecondes.

problème	**CD** $\vec{s}$	**MX** $\hookrightarrow_c$	**CDMX** $\vec{s}\hookrightarrow_c$	**CT** $\hookrightarrow_{\#c}$
*Telecom*3-5-5-32	68	0	3	31
*Telecom*3-5-6-40	105	0	7	57
*Telecom*3-5-6-64	408	1	15	170
*Telecom*4-5-4-33	110	1	1	37
*Telecom*4-5-5-43	324	1	7	145
*Telecom*4-6-5-56	489	3	8	321
*Telecom*4-6-6-63	273	0	7	186

Tab. 9.5 : Résultats pour l'utilisation de plusieurs instances du problème *Telecom*, obtenues avec « CHOCO *with a trace* ». Toutes les durées sont en millisecondes.

9.3.1 Simulation de l'utilisation d'une relation de transition

Commençons par les relations de transition, c'est-à-dire les instances de *Drone* et *ObsToMem*. Nous leur avons appliqué les quatre scénarios que nous avons déjà décrits en section 6.3.1. Le premier consiste à choisir une action exécutable dans l'état courant, et à récupérer un état parmi ceux auxquels elle peut conduire. Les résultats sont présentés tableau 9.2. Le scénario 2 est similaire, mais aucune action n'est choisie : on veut simplement connaître tous les états suivants possibles. Le tableau 9.3 contient les résultats pour cette simulation. Le scénario 3 vise à récupérer un état précédent l'état courant ; enfin, l'objectif du scénario 4 est de trouver un couple état-action qui peut mener à l'état courant. Les résultats pour les scénarios 3 et 4 sont présentés tableau 9.4.

Sans grande surprise, le profil général de chaque opération est similaire à ce que nous avons observé dans la section 6.3.1 : l'opération la plus longue est le premier conditionnement, suivi de l'oubli pour le scénario 3. L'extraction de modèle est rapide, quelle que soit la taille du graphe. Les opérations spéciales **CDMX** et **CDFOMX** donnent également de bons résultats. Il est cependant notable qu'alors qu'elles sont plus lentes que l'extraction de modèle dans les scénarios 1 et 2 (recherche en avant dans le système états-transitions), c'est le contraire dans les scénarios 3 et 4 (recherche en arrière).

En résumé, la manipulation de base de relations de transition compilées en *set-labeled diagrams* semble possible en pratique ; il serait intéressant d'étudier si des utilisations plus complexes restent faisables, telles que l'exploitation d'un `FSDD` pour construire une politique de décision avec une approche de planification par *model-checking* [GT99].

9.3.2 Simulation de l'utilisation du *benchmark Telecom*

Le problème sur lequel porte le *benchmark Telecom* est similaire à celui traité par *ObsToMem*, mais ce dernier est représenté comme une relation de transition (« si ce COM est en panne, il ne peut plus être utilisé dans l'état suivant »), tandis

que *Telecom* est codé comme un ensemble de configurations valides. Une utilisation en ligne du réseau de contraintes est de spécifier un état observé du système — c'est-à-dire, tel canal d'entrée est actif ou non, tel amplificateur est en panne — et d'extraire une configuration valide, c'est-à-dire un chemin reliant chaque canal d'entrée actif à un amplificateur et un canal de sortie disponibles. Ce scénario combine une nouvelle fois les opérations de conditionnement et d'extraction de modèle. Pour ce *benchmark*, il peut également être intéressant de compter le nombre de configurations possibles, pour avoir par exemple une idée de l'état général de fonctionnement du système. Cela revient à la requête standard de comptage de modèles (**CT**). Nous présentons dans le tableau 9.5 des résultats expérimentaux pour ces deux scénarios.

Comme pour les relations de transition de la section précédente, les simulations du premier scénario sont satisfaisantes. Utiliser une telle forme compilée pour indiquer à un contrôleur en ligne quelles sont les configurations valides semble une idée applicable en pratique, surtout en s'appuyant sur la requête dédiée **CDMX**, qui est plus rapide de plusieurs ordres de grandeur que la combinaison de **CD** et **MX**. Compter les configurations valides s'avère également une tâche faisable en ligne.

Conclusion

Dans cette thèse, nous nous sommes penchés sur l'application de la compilation de connaissances au contrôle de systèmes autonomes aéronautiques et spatiaux. Nous n'avons trouvé dans la littérature aucune étude sur la compilation de problèmes impliquant des variables continues, alors même que ces problèmes sont courants lorsque l'on s'intéresse à des systèmes autonomes réels. Notre approche a donc consisté à définir et étudier des langages-cibles de compilation :

- permettant de représenter des fonctions booléennes à la fois sur des variables continues et discrètes ;
- supportant en temps polynomial les requêtes et transformations que nous avons identifiées comme étant fondamentales pour la planification basée sur la compilation ;
- aussi compacts que possible — et donc aussi généraux que possible.

Cette conclusion résume les contributions et les limitations de notre étude, et donne des pistes pour des travaux futurs.

Contributions

État de l'art de la carte de compilation

Nous avons étendu le cadre existant autour de la carte de compilation [GK+95, DM02, FM07, FM09] de façon à ce qu'il inclue des langages de représentation pour n'importe quel type de fonction. En particulier, nous avons défini un langage général, qui englobe tous les langages basés sur des graphes permettant de représenter des fonctions booléennes, et qui nous permet d'étendre de manière simple les langages existants des variables booléennes aux variables discrète ou continues — notre cadre définit ainsi, par exemple, le langage `MDD` comme un sous-langage

particulier d'OBDD. Nous avons également proposé une extension aux variables non-booléennes des requêtes et transformations classiques sur les langages booléens. En rassemblant dans notre cadre étendu les résultats connus, nous avons dressé la carte de compilation des langages booléens basés sur des graphes — qui reste cependant encore incomplète.

Compilation de connaissances pour la planification

Nous avons étudié la littérature concernant les applications de la compilation de connaissances aux problèmes de planification, et donné un aperçu de ce qui a déjà été fait. Nous avons proposé de nouvelles requêtes et transformations utiles en planification — mais aussi dans des applications de diagnostic et de configuration. Nous avons identifié les requêtes et transformations importantes qu'un langage-cible de compilation doit satisfaire pour pouvoir être utilisé en ligne par un système autonome. Nous n'avons pas trouvé de travaux concernant la compilation de problèmes portant sur des variables continues, et avons décidé d'étudier de nouveaux langages-cibles adaptés à cet objectif.

Un langage-cible pour les variables réelles

Nous avons défini le langage des *automates à intervalles* (IAs), qui généralisent les diagrammes de décision binaires de plusieurs manières :

- les variables peuvent être discrètes ou continues ;
- les arcs sont étiquetés par des intervalles et non des singletons ;
- les nœuds de décision ne sont pas exclusifs, c'est-à-dire que les étiquettes des arcs ne sont pas nécessairement disjointes ;
- les nœuds purement disjonctifs sont autorisés.

Cette généralisation a nécessité une modification de la procédure classique de *réduction* des diagrammes de décision binaires.

Nous avons également identifié un sous-langage de IA, celui des automates à intervalles *convergents*, qui satisfait plusieurs requêtes et transformations importantes, parmi lesquelles **CO**, **MX**, **CD** et **FO**, ce qui en fait un bon langage-cible pour la planification en ligne.

Après avoir dressé la carte de compilation des IAs et des FIAs, comprenant les résultats de compacité et de satisfaction de requêtes et de transformations, nous avons décrit des méthodes permettant de compiler des problèmes vers FIA, notamment en *traçant* l'arbre de recherche d'un solveur de contraintes basé sur les intervalles, dans le style de « DPLL *with a trace* » [HD05].

Expérimentation des FIAs

Nous avons développé une bibliothèque de manipulation d'automates à intervalles, incluant une table de nœuds uniques pour traiter les nœuds isomorphes, ainsi que diverses requêtes et transformations, notamment celles utilisées pour contrôler des systèmes autonomes. Nous avons également implémenté un prototype du compilateur « RealPaver *with a trace* ». Nos premières expériences ont montré que l'utilisation pratique des FIAs pour le contrôle de systèmes autonomes est envisageable : les formes compilées pour nos *benchmarks* sont de taille raisonnable, ce qui pourrait leur permettre d'être embarqués ; et l'exécution des opérations, bien que non encore optimisées, est assez rapide pour qu'une utilisation en ligne soit possible.

Néanmoins, le gain en espace mémoire en comparaison aux diagrammes de décision binaires — grâce à l'absence de discrétisation, nécessaire sur des variables booléennes — est vraisemblablement compensé par le fait que les nombres réels prennent plus de place en mémoire que les entiers. Cependant, nous avons remarqué que les automates à intervalles définissent une discrétisation intrinsèque des domaines des variables, ce qui signifie qu'il est possible de traduire les IAs vers un langage-cible sur des variables *discrètes*, sans perte d'information et sans augmenter la taille de la forme compilée. Nous avons par conséquent décidé d'étudier la famille de langages correspondant aux IAs discrets.

Un nouveau langage-cible pour les variables discrètes

Nous avons défini le langage des *set-labeled diagrams* (SDs), qui généralisent les diagrammes de décision binaires de plusieurs manières :

- les variables sont discrètes, mais peuvent avoir un domaine non borné ;
- les arcs sont étiquetés par des unions d'intervalles, et non des singletons ;
- les nœuds de décision ne sont pas exclusifs, c'est-à-dire que les étiquettes des arcs ne sont pas nécessairement disjointes ;
- les nœuds purement disjonctifs sont autorisés.

Une nouvelle fois, nous avons adapté la procédure de réduction à ces nouvelles spécifications, et identifié des sous-langages de `SD`, parmi lesquels ceux des SDs *convergents* (`FSD`), des SDs *ordonnés* (`OSD`), et leurs équivalents satisfaisant la *décision exclusive* : `SDD`, `FSDD` et `OSDD`.

Après avoir démontré de manière formelle que les FIAs peuvent être transformés en FSDs de taille similaire en temps polynomial, nous avons dressé la carte de compilation de tous les langages de la famille de `SD`, avec leur compacité relative et les requêtes et transformations qu'ils supportent en temps polynomial. En particulier, `FSD` satisfait les requêtes et transformations supportées par `FIA`, ce qui n'est pas surprenant, mais garantit que la manipulation en ligne de FIAs transformés en

FSDs n'est pas plus difficile que la manipulation directe des FIAs. FSDD et OSDD peuvent également s'avérer intéressants pour certaines applications, en fonction des requêtes et transformations nécessaires.

Nous avons décrit une méthode de compilation de problèmes en FSDD ou OSDD, adaptée de l'approche « DPLL *with a trace* » [HD05], qui consiste à s'appuyer sur la trace d'exécution d'un solveur de contraintes. En faisant varier, dans le solveur, les paramètres de sélection de variables et de branchement sur le domaine, on peut obtenir des formes compilées ayant diverses propriétés.

Expérimentation des FSDs

Nous avons écrit un paquetage permettant de manipuler tous les langages de la famille de SD, comprenant la plupart des opérations polynomiales de la carte. Nous avons implémenté le compilateur « CHOCO *with a trace* », qui est capable de construire des FSDDs ou des OSDDs, selon le choix de l'utilisateur. Nos premières expériences sont prometteuses : passer des FIAs aux FSDs permet de gagner de l'espace sans augmenter la durée des opérations. Qui plus est, la compilation de problèmes discrets en FSDDs ou OSDDs semble praticable, et la manipulation des formes compilées en ligne n'est pas problématique a priori. Il est à noter que nous n'avons dans cette thèse exploré qu'une petite partie des possibilités de « CHOCO *with a trace* » ; ainsi, nous n'avons par exemple pas montré l'impact des heuristiques de choix de variable et de partage de domaines sur la taille des formes compilées.

Perspectives

Il reste beaucoup de travail à effectuer sur « CHOCO *with a trace* ». Notre bibliothèque permet de construire des FSDDs avec répétition de variables et sans ordre statique ; utiliser des heuristiques pour choisir les variables ou pour diviser les domaines pourrait donc donner des résultats intéressants. Une possibilité serait d'essayer de maximiser l'utilisation du cache : cela peut se faire, par exemple, en regardant une étape plus loin avant de choisir la variable, de manière à déterminer celle qui minimise le nombre d'ouvertures de nœuds dans l'arbre de recherche. En passant, signalons que la question de la compilation de FSDs purs, non-déterministes, reste ouverte — comme c'est le cas pour les DNNFs purs.

Des expériences plus avancées sur la planification avec des IAs ou des SDs pourrait inclure la construction d'une politique (par exemple en utilisant un algorithme de planification forte par *model-checking* [GT99]) et des comparaisons avec d'autres langages de la littérature.

D'un point de vue théorique, nos travaux soulèvent également des questions sur les représentations : comme nous l'avons montré, tout FIA peut être transformé en temps polynomial en un FSD « équivalent ». Pour établir ce fait, nous avons dû introduire une notion d'équivalence spécifique, différente de l'équivalence classique

sur les fonctions booléennes. En effet, les variables dans les cadres `FIA` et `FSD` sont complètement différentes ; d'une certaine manière, l'équivalence que nous avons définie est une « équivalence modulo un changement de variables ». Dans de futurs travaux, nous nous pencherons sur ces relations particulières entre les langages de représentation.

Bibliographie

[Ake78] Sheldon B. Akers. « Binary Decision Diagrams ». Dans : *IEEE Transactions on Computers* 27.6 (1978), p. 509–516.

[Ami99] Jérôme Amilhastre. « Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes ». Thèse de doct. Université Montpellier II, 1999.

[AH97] Henrik Reif Andersen et Henrik Hulgaard. « Boolean Expression Diagrams ». Dans : *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS), Warsaw, Poland*. 1997, p. 88–98.

[SHB00] Robert St-Aubin, Jesse Hoey et Craig Boutilier. « APRICODD : Approximate Policy Construction Using Decision Diagrams ». Dans : *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS)*. 2000, p. 1089–1095.

[BF+97] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo et Fabio Somenzi. « Algebraic Decision Diagrams and Their Applications ». Dans : *Formal Methods in System Design* 10.2/3 (1997), p. 171–206.

[Bar03] Anthony Barrett. « Domain Compilation for Embedded Real-Time Planning ». Dans : *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2003.

[Bar77] Jon Barwise, éd. *Handbook of Mathematical Logic*. North-Holland, 1977.

[BVC07] Grégory Beaumet, Gérard Verfaillie et Marie-Claire Charmeau. « Estimation of the Minimal Duration of an Attitude Change for an Autonomous Agile Earth-Observing Satellite ». Dans : *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2007, p. 3–17.

[Bel57] Richard Bellman. « A Markovian Decision Process ». Dans : *Indiana University Mathematics Journal* 6 (4 1957), p. 679–684. issn : 0022-2518.

[BYD07] Salem Benferhat, Safa Yahi et Habiba Drias. « On the Compilation of Stratified Belief Bases under Linear and Possibilistic Logic Policies ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, p. 2425–2430.

[BG01] Blai Bonet et Hector Geffner. « Planning as Heuristic Search ». Dans : *Artificial Intelligence Journal* 129.1–2 (2001), p. 5–33.

[BG06] Blai Bonet et Hector Geffner. « Heuristics for Planning with Penalties and Rewards using Compiled Knowledge ». Dans : *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2006, p. 452–462.

[Bry86] Randall E. Bryant. « Graph-Based Algorithms for Boolean Function Manipulation ». Dans : *IEEE Transactions on Computers* 35.8 (1986), p. 677–691.

[CD97] Marco Cadoli et Francesco M. Donini. « A Survey on Knowledge Compilation ». Dans : *AI Communications* 10.3–4 (1997), p. 137–150.

[CGT03] Claudio Castellini, Enrico Giunchiglia et Armando Tacchella. « SAT-based Planning in Complex Domains : Concurrency, Constraints and Nondeterminism ». Dans : *Artificial Intelligence Journal* 147.1–2 (2003), p. 85–117.

[CHO10] CHOCO Team. *CHOCO : An Open Source Java Constraint Programming Library*. Research report 10-02-INFO. École des Mines de Nantes, 2010. url : http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf.

[CG+97] Alessandro Cimatti, Fausto Giunchiglia, Enrico Giunchiglia et Paolo Traverso. « Planning via Model Checking : A Decision Procedure for $\mathcal{AR}$ ». Dans : *Proceedings of the European Conference on Planning (ECP)*. 1997, p. 130–142.

[CRT98a] Alessandro Cimatti, Marco Roveri et Paolo Traverso. « Automatic OBDD-Based Generation of Universal Plans in Non-Deterministic Domains ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, p. 875–881.

[CRT98b] Alessandro Cimatti, Marco Roveri et Paolo Traverso. « Strong Planning in Non-Deterministic Domains Via Model Checking ». Dans : *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 1998, p. 36–43.

[CGP99] Edmund M. Clarke, Orna Grumberg et Doron Peled. *Model Checking*. MIT Press, 1999. isbn : 9780262032704.

[CL+01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press et McGraw-Hill Book Company, 2001. isbn : 0-262-03293-7, 0-07-013151-1.

[CK+07] William Cushing, Subbarao Kambhampati, Mausam et Daniel S. Weld. « When is Temporal Planning Really Temporal ? » Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, p. 1852–1859.

[DLPT02] Ugo Dal Lago, Marco Pistore et Paolo Traverso. « Planning with a Language for Extended Goals ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2002, p. 447–454.

[Dar98] Adnan Darwiche. « Model-Based Diagnosis Using Structured System Descriptions ». Dans : *Journal of Artificial Intelligence Research (JAIR)* 8 (1998), p. 165–222.

[Dar01a] Adnan Darwiche. « Decomposable Negation Normal Form ». Dans : *Journal of the ACM* 48.4 (2001), p. 608–647. issn : 0004-5411.

[Dar01b] Adnan Darwiche. « On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision ». Dans : *Journal of Applied Non-Classical Logics* 11.1–2 (2001), p. 11–34.

[DM02] Adnan Darwiche et Pierre Marquis. « A Knowledge Compilation Map ». Dans : *Journal of Artificial Intelligence Research (JAIR)* 17 (2002), p. 229–264.

[DM04] Adnan Darwiche et Pierre Marquis. « Compiling Propositional Weighted Bases ». Dans : *Artificial Intelligence Journal* 157.1–2 (2004), p. 81–113.

[DLL62] Martin Davis, George Logemann et Donald W. Loveland. « A Machine Program for Theorem-Proving ». Dans : *Communications of the ACM* 5.7 (1962), p. 394–397.

[DHW94] Denise Draper, Steve Hanks et Daniel S. Weld. « Probabilistic Planning with Information Gathering and Contingent Execution ». Dans : *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 1994, p. 31–36.

[Eme90] E. Allen Emerson. « Temporal and Modal Logic ». Dans : *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*. 1990, p. 995–1072.

[EHN96] Kutluhan Erol, James A. Hendler et Dana S. Nau. « Complexity Results for HTN Planning ». Dans : *Annals of Mathematics and Artificial Intelligence* 18.1 (1996), p. 69–93.

[FM08a] Hélène Fargier et Pierre Marquis. « Extending the Knowledge Compilation Map : Closure Principles ». Dans : *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 2008, p. 50–54.

[FM08b] Hélène Fargier et Pierre Marquis. « Extending the Knowledge Compilation Map : Krom, Horn, Affine and Beyond ». Dans : *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2008, p. 442–447.

[FM09] Hélène Fargier et Pierre Marquis. « Knowledge Compilation Properties of Trees-of-BDDs, Revisited ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2009, p. 772–777.

[FV04] Hélène Fargier et Marie-Catherine Vilarem. « Compiling CSPs into Tree-Driven Automata for Interactive Solving ». Dans : *Constraints* 9.4 (2004), p. 263–287.

[FM06] Hélène Fargier et Pierre Marquis. « On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae ». Dans : *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2006.

[FM07] Hélène Fargier et Pierre Marquis. « On Valued Negation Normal Form Formulas ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, p. 360–365.

[FG00] Paolo Ferraris et Enrico Giunchiglia. « Planning as Satisfiability in Nondeterministic Domains ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2000, p. 748–753.

[FN71] Richard Fikes et Nils J. Nilsson. « STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1971, p. 608–620.

[FL06] Maria Fox et Derek Long. « Modelling Mixed Discrete-Continuous Domains for Planning ». Dans : *Journal of Artificial Intelligence Research (JAIR)* 27 (2006), p. 235–297.

[Gef11] Hector Geffner. *Advanced Introduction to Planning : Models and Methods*. Tutorial at the International Joint Conference on Artificial Intelligence (IJCAI). 2011. url : http://www.dtic.upf.edu/~hgeffner/tutorial-planning-ijcai-2011.html.

[GM94] Jordan Gergov et Christoph Meinel. « Efficient Boolean Manipulation with OBDD's Can Be Extended to FBDD's ». Dans : *IEEE Transactions on Computers* 43.10 (1994), p. 1197–1209.

[GNT04] Malik Ghallab, Dana Nau et Paolo Traverso. *Automated Planning : Theory & Practice*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004. isbn : 1558608567.

[Gib85] Alan M. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985. isbn : 9780521288811.

[GMS98] Enrico Giunchiglia, Alessandro Massarotto et Roberto Sebastiani. « Act, and the Rest Will Follow : Exploiting Determinism in Planning as Satisfiability ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, p. 948–953.

[GT99] Fausto Giunchiglia et Paolo Traverso. « Planning as Model Checking ». Dans : *Proceedings of the European Conference on Planning (ECP)*. 1999, p. 1–20.

[GK+95] Goran Gogic, Henry A. Kautz, Christos H. Papadimitriou et Bart Selman. « The Comparative Linguistics of Knowledge Representation ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1995, p. 862–869.

[GB06] Laurent Granvilliers et Frédéric Benhamou. « RealPaver : an Interval Solver Using Constraint Satisfaction Techniques ». Dans : *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), p. 138–156.

[HH+08] Tarik Hadzic, John N. Hooker, Barry O'Sullivan et Peter Tiedemann. « Approximate Compilation of Constraints into Multivalued Decision Diagrams ». Dans : *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2008, p. 448–462.

[HG00] Patrik Haslum et Hector Geffner. « Admissible Heuristics for Optimal Planning ». Dans : *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*. 2000, p. 140–149.

[HS+99] Jesse Hoey, Robert St-Aubin, Alan J. Hu et Craig Boutilier. « SPUDD : Stochastic Planning Using Decision Diagrams ». Dans : *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*. 1999, p. 279–288.

[HN01] Jörg Hoffmann et Bernhard Nebel. « The FF Planning System : Fast Plan Generation Through Heuristic Search ». Dans : *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), p. 253–302.

[HD04] Jinbo Huang et Adnan Darwiche. « Using DPLL for Efficient OBDD Construction ». Dans : *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2004, p. 157–172.

[HD05] Jinbo Huang et Adnan Darwiche. « DPLL with a Trace : From SAT to Knowledge Compilation ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, p. 156–162.

[JKK09] Saket Joshi, Kristian Kersting et Roni Khardon. « Generalized First Order Decision Diagrams for First Order Markov Decision Processes ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2009, p. 1916–1921.

[KMS96] Henry A. Kautz, David A. McAllester et Bart Selman. « Encoding Plans in Propositional Logic ». Dans : *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1996, p. 374–384.

[KS92a] Henry A. Kautz et Bart Selman. « Forming Concepts for Fast Inference ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. San Jose (CA), 1992, p. 786–793.

[KS92b] Henry A. Kautz et Bart Selman. « Planning as Satisfiability ». Dans : *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 1992, p. 359–363.

[Kov11] Dániel L. Kovács. *Complete BNF Description of PDDL 3.1*. 2011. url : http://www.plg.inf.uc3m.es/ipc2011-deterministic/Resources?action=AttachFile&do=get&target=kovacs-pddl-3.1-2011.pdf.

[LS+07] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary et Vincent Vidal. « Transposition Tables for Constraint Satisfaction ». Dans : *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2007, p. 243–248.

[LS06] Christophe Lecoutre et Radoslaw Szymanek. « Generalized Arc Consistency for Positive Table Constraints ». Dans : *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2006, p. 284–298.

[Lee59] C. Y. Lee. « Representation of Switching Circuits by Binary Decision Programs ». Dans : *Bell Systems Technical Journal* 38 (1959), p. 985–999.

[Mar08] Pierre Marquis. *Knowledge Compilation : A Sightseeing Tour*. Tutorial at the European Conference on Artificial Intelligence. 2008. url : http://www.cril.univ-artois.fr/~marquis/tutorialNotes-ECAI08-PMarquis.pdf.

[MD06] Robert Mateescu et Rina Dechter. « Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs) ». Dans : *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2006, p. 329–343.

[MDM08] Robert Mateescu, Rina Dechter et Radu Marinescu. « AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models ». Dans : *Journal of Artificial Intelligence Research (JAIR)* 33 (2008), p. 465–519.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993, p. I–XV, 1–194. isbn : 978-0-7923-9380-1.

[OP06] Barry O'Sullivan et Gregory M. Provan. « Approximate Compilation for Embedded Model-Based Reasoning ». Dans : *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2006.

[PB+05] Héctor Palacios, Blai Bonet, Adnan Darwiche et Hector Geffner. « Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations ». Dans : *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2005, p. 141–150.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994, p. I–XV, 1–523. isbn : 978-0-201-53082-7.

[PT01] Marco Pistore et Paolo Traverso. « Planning as Model Checking for Extended Goals in Non-deterministic Domains ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2001, p. 479–486.

[PV+10] Cédric Pralet, Gérard Verfaillie, Michel Lemaître et Guillaume Infantès. « Controller Synthesis for Autonomous Systems : A Constraint-Based Approach ». Dans : *Proceedings of the 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS 2010)*. Sous la dir. de Takashi Kubota. 2010.

[Rin11] Jussi Rintanen. *Algorithms for Classical Planning*. Tutorial at the International Joint Conference of Artificial Intelligence (IJCAI). 2011. url : http://users.cecs.anu.edu.au/~jussi/ijcai11tutorial/.

[Sac75] Earl D. Sacerdoti. « The Nonlinear Nature of Plans ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1975, p. 206–214.

[SF96] Djamila Sam-Haroud et Boi Faltings. « Consistency Techniques for Continuous Constraints ». Dans : *Constraints* 1.1/2 (1996), p. 85–118.

[Seb07] Roberto Sebastiani. « Lazy Satisability Modulo Theories ». Dans : *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 3.3–4 (2007), p. 141–224.

[SFJ00] David E. Smith, Jeremy Frank et Ari K. Jónsson. « Bridging the Gap between Planning and Scheduling ». Dans : *The Knowledge Engineering Review* 15 (1 mar. 2000), p. 47–83. issn : 0269-8889. doi : 10.1017/S0269888900001089. url : http://dl.acm.org/citation.cfm?id=975748.975752.

[SW98] David E. Smith et Daniel S. Weld. « Conformant Graphplan ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1998, p. 889–896.

[Som05] Fabio Somenzi. *CUDD : Colorado University Decision Diagram package, release 2.4.1*. 2005. url : http://vlsi.colorado.edu/~fabio/CUDD/.

[SK+90] Arvind Srinivasan, Timothy Kam, Sharad Malik et Robert K. Brayton. « Algorithms for Discrete Function Manipulation ». Dans : *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. Nov. 1990, p. 92–95.

[ST98] Karsten Strehl et Lothar Thiele. « Symbolic Model Checking of Process Networks Using Interval Diagram Techniques ». Dans : *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. 1998, p. 686–692.

[Vem92] Nageshwara Rao Vempaty. « Solving Constraint Satisfaction Problems Using Finite State Automata ». Dans : *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1992, p. 453–458.

[VP08a] Alberto Venturini et Gregory Provan. « Incremental Algorithms for Approximate Compilation ». Dans : *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2008, p. 1495–1499.

[VP08b] Gérard Verfaillie et Cédric Pralet. « Utiliser des chronogrammes pour modéliser des problèmes de planification et d'ordonnancement ». Dans : *Proceedings of the French Conference on Planning, Decision, and Learning (JFPDA, Journées Francophones de Planification, Décision et Apprentissage)*. 2008.

[WH06] Michael Wachter et Rolf Haenni. « Propositional DAGs : A New Graph-Based Language for Representing Boolean Functions ». Dans : *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 2006, p. 277–285.

[Wal00] Toby Walsh. « SAT $\vee$ CSP ». Dans : *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. 2000, p. 441–456.

[WFD07] Robert Wille, Görschwin Fey et Rolf Drechsler. « Building Free Binary Decision Diagrams Using SAT Solvers ». Dans : *Facta Universitatis, Series : Electronics and Energetics* 20.3 (2007), p. 381–394. issn : 03533670.

[Wil05] Nic Wilson. « Decision Diagrams for the Computation of Semiring Valuations ». Dans : *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2005, p. 331–336.

Index des symboles