Université Paris Diderot - Paris 7
Laboratoire d'Informatique Algorithmique : Fondements et Applications

Thèse
pour l'obtention du diplôme de
Docteur de l'Université Paris Diderot,
spécialité Informatique

à l'École Doctorale de Sciences Mathématiques de Paris Centre

Analyse de XML avec données non-bornées

présentée et soutenue publiquement par

Claire David

le 7 Mai 2009

Directeurs de thèse :	Anca Muscholl	
	Luc Segoufin	
Après avis de :	Nicole Bidoit	
	Stéphane Demri	
Jury composé de :	Nicole Bidoit	Rapporteur
	Ahmed Bouajjani	
	Christian Choffrut	
	Stéphane Demri	Rapporteur
	Anca Muscholl	Directrice
	Luc Segoufin	Directeur

Merci à tous !

Mes premiers contacts avec la recherche sont prénatals et les heureuses rencontres que j'ai faites depuis ma découverte de l'informatique théorique à l'ENS Cachan sont trop nombreuses pour être toutes citées ici. Je remercierai tout de même Hubert et Catherine ainsi que tous les membres de la non-équipe pédagogique de l'époque qui ont donné naissance au département d'informatique actuel.

Merci à mes deux directeurs de thèse qui ont pris leur rôle très à cœur et ont toujours été là pour me conseiller. Merci de m'avoir proposé mon sujet de thèse et d'avoir orienté ma recherche vers des problèmes que je trouve passionnants. Un grand merci à Anca pour son enseignement et ses conseils avant même mon premier stage de recherche en licence. Ce fut et c'est toujours un grand plaisir de travailler avec toi. Merci aussi à Luc de m'avoir fait découvrir le domaine des bases de données dont je n'avais pas imaginé toutes les richesses. Merci pour tous les bons moments de recherche et l'aide qu'il m'a apportée lorsque j'étais bloquée. Merci aussi pour ta patience, ta délicatesse et ton soutien dans mes moments de faiblesse.

Un grand merci à mes rapporteurs Nicole et Stéphane pour leur lecture approfondie et détaillée ainsi que leurs commentaires pertinents.
Je voudrais également remercier Stéphane pour l'intérêt qu'il a toujours porté à mes travaux et les nombreuses discussions qui en ont découlé. Merci pour tes propres travaux sur lesquels j'aurais eu grand plaisir à travailler moi-même. Mon regret que nous n'ayons jamais eu plus de temps pour travailler ensemble est compensé par le plaisir que tu aies pu être rapporteur de ma thèse.

Un grand merci à Ahmed et Christian pour avoir accepté de faire partie de mon jury et pour avoir montré de l'intérêt pour mes travaux durant ma thèse.

Merci à tous les thésards du LIAFA, de PPS et d'ailleurs avec qui j'ai passé de très bons moments et sans qui ces quatre années n'auraient pas été les mêmes. Vous être trop nombreux pour tous être cités : merci à Laura, Antoine, Benjamin, Manu, Sam, Olivier, Pierre(s), Mathilde, Vincent, Cezara, Grégoire, Medhi, Faouzi, Boris, Simon, Basile, Stéphane(s), Christine, Barbara, Séverine, Amélie, Jérémie, Allessandro, Eude et tous les autres.

Merci aux organisateurs successifs du séminaire thésards.

Merci à Pierre pour tes pitoyables performances de basketteur qui m'ont toujours amusée. Merci aussi pour le "canapé du Docteur Moro" qui depuis ton départ est le lieu de repos des journées trop longues ou des nuits trop courtes.

Merci à Florian, Grégoire et Julien pour s'être relayés à mes côtés pendant la phase alpine de ma rédaction.

Merci à Mikołaj pour les discussions de recherche que nous avons eu pendant ton post-doc puis après lors de nos rencontres. Indépendamment du côté scientifique, j'ai toujours eu plaisir à discuter avec toi ou chercher le plus haut point de vue des différentes villes que nous avons visitées ensemble.

Merci à Mathilde pour avoir enduré la preuve de $\mathrm{FO}^2(\sim, +1)$ sur les mots. Si cette notation vous rebute, ce n'est rien comparé à la réalité de cette preuve.

Merci à Henrik et Wouter sans qui je serais morte de faim en Pologne.

Un grand merci à Julien pour les promenades au pas de course à n'importe quelle heure du jour et de la nuit. Merci infiniment pour t'être brûlé les yeux à la relecture de ma thèse tant dans ses versions pré-préliminaires que dans sa version finale.

Merci à Blaise mon "grand frère" et ami depuis nos premières parties de badminton à Cachan, sur qui je peux et j'ai toujours pu compter.

Merci infiniment Florian pour ton amitié, ton soutien, tes massages, tes relectures et pour toutes ces heures que tu as passées à jouer le catalyseur de recherche dans le sous-marin. Sans toi, cette thèse ne serait pas ce qu'elle est. Merci pour ta disponibilité et tout ce que je n'ai pas cité ici.

Merci à toutes les personnes avec qui j'ai eu le plaisir d'enseigner au cours de mon monitorat : Manu, Sam, Antoine, Ines, Eugène, Christophe, Florian, Guillaume, Grégoire...

Merci aux ingénieurs systèmes qui ont réussi, à une époque où ce n'était qu'un rêve, à m'acheter un ordinateur portable vierge de tout système d'exploitation.

Un grand merci à Noëlle pour son immense soutien à tous les thésards et sans qui le LIAFA n'existerait pas. Merci à Michèle qui gère les thésards au niveau de l'école doctorale.

Merci à tous les chercheurs de Chevaleret pour leur compagnie et les discussions (d'ordre scientifique ou autre) autour de la table à café ou au détour d'un couloir. Merci à Jean, Juliusz, Olivier(s), Mihaela, Dominique, Jean-Éric, Irène, Arnaud, Christian, Peter, Sylvain, Isabelle, Roberto, Anne, Nicolas, Luc, François, Eugène...

Un grand merci à tous les chercheurs que j'ai rencontrés lors de mes/leurs voyages. Merci à Dietmar grâce à qui une soirée foot à Cambridge s'est transformée en la première de nos longues conversations. Merci à Jacques, Sophie(s), Christof, Nicole, Balder, Wim, Marc, Patricia, Serge, Victor, Ranko...

Merci à l'équipe d'Edinburgh qui m'a accueillie chaleureusement et avec qui j'ai plaisir à travailler lorsque l'on arrive à se retrouver. Merci en particulier à Filip, Leonid, Floris et Heiko.

Un grand merci à Florian pour avoir inventé la tradition du gâteau du vendredi qui contamine doucement les labos de France et d'Europe. Merci Mathilde d'avoir joué le jeu et fait le premier gâteau de cette longue série. Merci Eugène pour ton excellent tiramisu (indépendamment du fait que ce soit ton premier). Merci Laurent pour avoir eu le courage de faire ton premier gâteau ici durant ton post-doc. Merci Ahmed, je ne regrette pas d'avoir attendu plus d'un an et demi pour que tu te décides à cuisiner. Et merci à tous les cuisiniers du vendredi.

Un grand merci enfin à ma grande famille et mes amis pour tous les bons moments passés ensemble et votre soutien. Merci en particulier à mes parents pour avoir su vous adapter à mon rythme de vie quantique pendant ma rédaction. Merci de votre soutien et de votre disponibilité. Merci aussi à mon frère, ma soeur (qui a fini par réussir à me doubler en soutenant avant moi), à mes grand-parents et à ma tante Souris. Merci aussi à Aurélien et sa famille. Merci à Florian (l'autre), Damien, Anne et Katrin.

Et merci à tous ceux que j'ai pu oublier !

Table des matières

Chapitre 1

Introduction

Les mots et les arbres sont des structures abondamment utilisées dans des domaines variés de l'informatique. On mentionne ici deux exemples particuliers. Dans le domaine des bases de données semi-structurées, la structure d'un document XML correspond essentiellement à un arbre et on peut voir le chemin de la racine du document à une donnée textuelle comme un mot. Dans le domaine de la vérification et spécification de systèmes, des structures linéaires ou arborescentes sont utilisées pour représenter des comportements ou encore des configurations d'un système.

Dans ces deux domaines, on a besoin de décrire et manipuler des ensembles de structures. En effet dans le cadre d'une application donnée, on veut par exemple se restreindre à l'ensemble des documents XML représentant des données dans un certain format. De la même manière on veut pouvoir raisonner sur l'ensemble des comportements ou des configurations d'un système donné afin de pouvoir tester s'il vérifie une propriété. Dans le cas de XML, on parle de schéma ; dans le cas de la vérification, on parle de spécification du système. Dans les deux cas, il s'agit de se restreindre à un ensemble d'objets en fonction des caractéristiques du contexte dans lequel on se place. On a donc besoin de formalismes pour décrire ces ensembles (ou langages) de mots et arbres. De plus on veut savoir manipuler ces ensembles afin de pouvoir répondre à des questions du type :

- Une spécification est-elle cohérente ? Ou au contraire impossible à satisfaire ? (vide du langage correspondant)
- Deux spécifications sont-elles équivalentes ? (égalité de langage)
- Une spécification est-elle incluse dans une autre spécification ? (inclusion de langage)
- Une structure vérifie-t-elle la spécification ? (appartenance à un langage)

Le choix d'un formalisme de spécification se fait en étudiant conjointement son expressivité et la décidabilité ainsi que la complexité des problèmes énoncés ci-dessus.

Une manière classique de décrire un ensemble de mots ou d'arbres est d'utiliser un automate ou une formule logique. Un langage peut alors être vu comme l'ensemble des mots reconnus par l'automate ou encore comme l'ensemble des modèles de la formule. Dans le cas des mots et arbres étiquetés par un alphabet fini, l'étude de ces formalismes a fait l'objet de nombreux travaux. Il existe de nombreuses correspondances entre logiques et automates, l'exemple canonique étant l'équivalence entre la logique monadique du second ordre et les automates finis. Ces travaux se sont aussi attachés à développer des formalismes alternatifs ou étendus dont l'expressivité et la syntaxe varient en fonction des différents besoins (par exemple les logiques temporelles ou les automates à pile). L'objectif de ce travail a aussi été d'obtenir des formalismes robustes que l'on puisse manipuler efficacement. Les problèmes typiques auxquels on s'intéresse sont le vide, l'équivalence, l'inclusion entre langages ou simplement l'appartenance d'un objet à un langage.

Tous ces travaux concernent des langages de mots ou d'arbres étiquetés par un alphabet fini. Cela revient dans le cas des documents XML à faire abstraction des données et à ne considérer que la structure des documents qui peut être décrite par un ensemble fini de balises. Dans le cas de la vérification et spécification de système, cela correspond à abstraire le système en un modèle possédant un nombre fini d'états de contrôle.

Cependant, dans le cadre de certaines applications, on a besoin de représenter et de manipuler des données d'un domaine a priori infini. Dans le cadre des documents XML, les données sont naturellement présentes (texte/attributs) et constituent une composante essentielle du système. De la même manière, on veut pouvoir parler des valeurs des variables entières ou des adresses mémoire utilisées par un programme ou encore des identifiants des composantes d'un système distribué. On s'intéresse ici à une extension des structures linéaires et arborescentes "classiques" en considérant des mots et des arbres bi-étiquetés d'une part par une lettre d'un alphabet fini et d'autre part par ce qu'on appellera une donnée provenant d'un domaine a priori infini. On parle alors de mots et d'arbres de données. L'objectif de cette extension est de pouvoir représenter les données présentes dans les documents XML ou encore d'inclure les données dans la représentation de configurations ou comportements de systèmes par ce type de structure.

Dans ce contexte la question des formalismes de spécification doit être reconsidérée. Une approche naturelle est d'étendre les formalismes "classiques" utilisés pour décrire des langages de mots et d'arbres étiquetés sur un alphabet fini. Cette approche est délicate pour plusieurs raisons. Du point de vue de l'expressivité, cela dépend du domaine d'application dans lequel on se place et en particulier du type de données que l'on veut représenter. Un ensemble de données représentant les valeurs de variables réelles dans un programme de calcul a une structure différente d'un ensemble de données correspondant à des identifiants dans une base de données ou encore à des adresses mémoire d'un disque dur. Dans le premier cas, on a besoin de la relation d'ordre ou encore d'opérations arithmétiques, dans les autres cas, on peut imaginer ne disposer que de la relation d'égalité entre les données. Dans tous les cas l'indécidabilité des problèmes relatifs à la manipulation des langages constitue le principal problème auquel on est confronté.

Munir l'ensemble des données d'une relation d'égalité est une abstraction minimale que l'on veut considérer. Lors de la description de schémas pour les documents XML, un exemple classique de contrainte que l'on veut pouvoir exprimer est celui des contraintes dites de clé. Une clé permet d'identifier de manière unique un élément du document. Par exemple si un document XML représente le contenu d'une bibliothèque, on veut pouvoir exprimer le fait que les numéros ISBN de chacun de ses livres sont différents. Mais on peut aussi imaginer d'autres contraintes basées sur des tests d'égalité et différence comme par exemple imposer que deux numéros différents d'une même revue ont forcément des dates de parution différentes.

De la même manière cette abstraction des données est dans certains cas suffisante pour la spécification de systèmes ; on pense par exemple aux identifiants des processus d'un système distribué ou encore aux adresses mémoire utilisées par un programme. Lorsque l'on raisonne sur les adresses mémoire manipulées par un programme complexe, l'égalité suffit pour exprimer le fait que deux parties du programme n'écrivent pas au même endroit.

Malheureusement même en considérant l'ensemble des données muni de la simple opération d'égalité, la définition et la manipulation de formalismes de spécification posent déjà des problèmes. Si l'on cherche à étendre les formalismes classiques (logique et automates) on bascule très rapidement de la décidabilité à l'indécidabilité. Par exemple si l'on munit la logique monadique du second ordre d'un prédicat d'égalité sur les données, le problème de satisfaisabilité devient indécidable même dans le cas des mots de données. Néanmoins, on montrera dans cette thèse qu'il existe des situations dans lesquelles on parvient à décrire des formalismes expressifs et décidables.

Pour cela, il est nécessaire de faire des compromis entre les questions d'expressivité et de décidabilité. Dans cette thèse on s'intéresse à cette question de spécification de langages de mots et d'arbres de données. Afin de restreindre les problèmes d'expressivité a priori très variés selon le cadre d'application, on considère d'une part un ensemble de données muni de la seule opération d'égalité et d'autre part, on se focalise sur les questions d'expressivité posées par les documents XML.

Objectifs et contributions.

Le travail de cette thèse est motivé par la question de la gestion des données dans la définition et manipulation de formalismes de spécification de documents XML. Comme on l'a mentionné précédemment, on considère des versions étendues des mots et des arbres appelés mots et arbres de données. Un mot de données est un mot (a priori fini) dont chaque position est étiquetée d'une part par une lettre d'un alphabet fini Σ et d'autre part par une donnée provenant d'un domaine potentiellement infini $\mathcal{D}$. Un arbre de données est défini de la même manière comme un arbre fini (a priori ordonné) d'arité non bornée bi-étiqueté par Σ et $\mathcal{D}$. L'ensemble des données $\mathcal{D}$ est muni d'une relation d'égalité. Les seules opérations (a priori) autorisées sur les données sont des tests d'égalité et de différence entre deux données.

L'objectif des travaux présentés est de définir des formalismes génériques décidables pour décrire des langages de mots ou d'arbres de données.

On propose pour cela deux approches. La première approche, détaillée dans la première partie du manuscrit, est basée sur la logique du premier ordre munie d'un prédicat binaire permettant de tester des égalités de données.

On étudie cette logique sur les mots de données. On s'intéresse au problème de satisfaisabilité : "étant donnée une formule de la logique, existe-t-il un modèle pour cette formule ?". Ce problème est indécidable en général mais décidable si on se restreint à l'ensemble des formules utilisant au plus deux variables. Ce résultat utilise une correspondance entre logique et automates. En effet, on définit une classe d'automates appelés automates à données et on montre que le langage décrit par une formule du fragment à deux variables peut être reconnu par un tel automate. De plus le vide de ces automates est décidable, ce qui permet de conclure. On s'intéresse à différentes extensions du fragment à deux variables, le but étant de préciser la frontière de décidabilité et de donner une caractérisation logique des automates à données (strictement plus expressifs que le fragment à deux variables). D'un point de vue de la complexité, le vide des automates, ainsi que le problème de satisfaisabilité du fragment à deux variables est équivalent au problème du vide des automates multicompteurs aussi connus sous le nom de VASS ("Vector Addition System with States"). Ce problème est équivalent en terme de complexité au problème d'accessibilité dans les réseaux de Petri. Ce problème est décidable mais sa complexité exacte est une question ouverte depuis longtemps. La meilleure borne inférieure est ExpSpace et l'on ne connaît pas de borne supérieure élémentaire. On considère deux fragments de la logique en limitant le pouvoir de comparaison navigationelle des formules, on obtient alors des bornes de complexité allant de NExpTime à 3NExpTime.

Une partie de ces résultats s'étend aux arbres. Il existe un fragment décidable de la logique à deux variables sur les arbres. On s'intéresse au lien de ce résultat avec la problématique de description de schémas pour les documents XML. Un document XML pouvant être vu comme un arbre de données, ce résultat entraîne la décidabilité de tout problème d'analyse statique exprimable comme un problème de satisfaisabilité d'une formule de cette logique. On remarque en particulier que l'on peut exprimer n'importe quel langage régulier d'arbres (étiquetés sur l'alphabet fini) mais aussi des contraintes d'intégrité de type clé unaire ou encore inclusion.

Cette approche logique est élégante et générique, malheureusement les complexités obtenues sont très élévées. On propose une seconde approche basée sur des objets appelés "patterns". Cette approche est détaillée dans la seconde partie de ce manuscrit. L'un des objectifs de cette approche est d'obtenir des résultats de complexité plus raisonnables tout en considérant un formalisme plus intuitif. On s'intéresse alors au problème du model checking (appartenance d'un object à un langage) en plus du problème de satisfaisabilité.

Les patterns sont des objets simples qui permettent à la fois d'exprimer des contraintes sur la structure finie du modèle mais aussi d'exprimer des contraintes d'égalité et de différence entre les données. Un pattern d'arbre est constitué d'un arbre étiqueté dont les arêtes sont marquées par la relation enfant ou descendant, et de contraintes d'égalité et de différences de données entre des nœuds de cet arbre. Un arbre de données satisfait un pattern s'il existe une fonction injective des nœuds du pattern d'arbre dans ceux de l'arbre de données qui respecte les contraintes d'étiquettes, de structure ainsi que les contraintes de données. Les patterns de mots se décrivent de la même façon en considérant un mot comme un arbre d'arité 1. Un pattern permet ainsi de décrire un motif que l'on veut voir apparaître dans le modèle (propriété existentielle). Considérer la négation d'un pattern permet à l'inverse d'interdire certains motifs dans le modèle (propriété universelle). De manière plus générale, on considère des combinaisons booléennes de patterns. L'expressivité de ces formules est incomparable avec celle des formules du fragment à deux variables de la logique du premier ordre proposé plus haut.

On s'intéresse à deux problèmes en particulier : la satisfaisabilité en présence de contrainte régulière et le model-checking.

Le problème de satisfaisabilité est indécidable en général ; on donne deux façons d'obtenir la décidabilité du problème. Une première façon de procéder est de restreindre les formules considérées ou l'expressivité des patterns. Une deuxième solution proposée est de borner la profondeur des modèles considérés. Cette restriction est naturelle si l'on revient à notre motivation initiale par le langage XML ; en effet la profondeur d'un ensemble de documents dans un contexte d'utilisation est souvent bornée par le schéma. Un ensemble de documents est ainsi en général de profondeur bornée et faible mais d'arité quelconque. Dans les deux cas, on obtient alors des résultats de décidabilité avec des complexités allant de NP-complet à 2ExpTime-complet.

Le problème du model checking est le suivant : "étant donnés une formule et un arbre de données, l'arbre satisfait-il la formule ?". Ce problème est décidable ; en effet, il n'y a qu'un nombre fini de solutions possibles pour qu'un pattern apparaisse dans un arbre fini. La question intéressante est donc celle de la complexité du problème. On montre que le problème de model checking est NP-complet si l'on ne considère qu'un seul pattern d'arbre. Dans le cas général ce problème est dans la classe [1] $\mathrm{P}^{\|\mathrm{NP}}$ et difficile pour toute classe BH_k de la hiérachie booléenne de NP (BH).

Si l'on regarde le cas restreint des mots de données on obtient des résultats similaires. La principale différence est la complexité du problème de satisfaibilité ; la structure linéaire des mots par rapport à la structure branchante des arbres permet d'économiser une exponentielle.

Enfin, on explique les liens entre ces résultats et la problématique de l'analyse statique de documents XML en comparant l'expressivité de ce formalisme ainsi que les résultats de complexité obtenus avec des résultats préexistants dans le domaine. On reviendra en particulier sur les questions de clés ainsi que sur le langage navigationnel XPath.

État de l'Art

La problématique des données non-bornées a connu un grand intérêt ces dernières années. Certains travaux présentés dans cette partie sont contemporains de cette thèse. Le but est de donner un aperçu des travaux reliés à cette question et en s'intéressant plus particulièrement aux langages de données. Nous avons classé les résultats selon trois grands axes : logique et automates, spécification et vérification, et bases de données semi-structurées. Cependant, les frontières entre ces domaines sont poreuses, et certaines contributions sont transverses.

Logique et automates

Dans le contexte classique de la théorie des langages, Autebert, Beauquier et Boasson ont étudié les langages de mots finis étiquetés par un alphabet infini d'un point de vue algébrique. Dans [ABB80], ils définissent différentes classes de langages à l'aide de grammaires et de propriétés algébriques. Ils étudient le comportement de ces classes par rapport aux problèmes de clôture et d'inclusion et s'intéressent à l'adaptation de théorèmes classiques de la théorie des langages.

[1] La classe $\mathrm{P}^{\|\mathrm{NP}}$ est la classe des langages reconnaissables en temps polynomial par une machine de Turing qui peut poser une série de requêtes en parallèle à un oracle NP.

Dans [CG08], Choffrut et Grigorieff s'intéressent aux relations sur les mots de données. L'objectif de ces travaux est d'étendre, au cas des alphabets infinis, les résultats de Eilenberg, Elgot et Shepherdson qui montrent les liens entre les relations sur les mots, les automates synchrones et la logique du premier ordre dans le cas des alphabets finis. Ils étendent ainsi la notion d'automate synchrone à n-bandes afin de décrire des relations d'arité n sur l'ensemble des mots étiquetés par un alphabet infini. Les transitions de l'automate sont conditionnées par des formules constituées de tests d'égalité et de comparaisons à une constante sur le n-uplet des étiquettes correspondant à la position courante de la tête de lecture. Ces travaux donnent aussi une caractérisation logique de ce type de relations. Si on autorise un trop grand pouvoir expressif à la logique, on se heurte à des problèmes d'indécidabilité.

Les travaux de Bès étendent les résultats précédents en raisonnant sur des classes d'automates synchrones dont les transitions sont définies par des formules du premier ordre (ou même du second ordre) sur les données [Bès08]. Il étudie aussi des logiques correspondant aux automates.

Comparativement aux travaux de cette thèse, ils ne limitent a priori pas les opérations sur les données au simple test d'égalité mais s'autorisent à considérer des domaines de données du type $(\mathbf{N}, +)$. D'un autre côté, les formalismes proposés n'autorisent pas de comparer les valeurs de données entre différentes positions d'un mot. L'auteur propose des idées pour étendre ses travaux afin de permettre des tests d'égalité entre différentes positions du mot tout en restant décidable. Ceci reste cependant une perspective de travail.

Les travaux sur les automates à registres présentés dans [KF94] sont à l'origine de nombreuses extensions de la théorie des automates aux alphabets infinis. On reviendra d'ailleurs sur ces automates dans la première partie de la thèse. Kaminski et Francez définissent une extension des automates finis unidirectionnels à base de registres. Les registres permettent de stocker l'étiquette de la position courante afin de pouvoir l'utiliser dans des tests d'égalité plus tard au cours de l'exécution. Le vide de ces automates est décidable. Les travaux présentés dans [NSV04] portent sur des extensions de ce modèle. Neven, Schwentick et Vianu donnent un aperçu de différents formalismes de spécification de langages de mots de données. Ils décrivent et comparent : le modèle des automates à jetons, le modèle des automates à registres (bidirectionnels), et une version étendue de la logique monadique du second ordre. Ces classes de langages sont en général indécidables. Dans cette thèse, on cherche à definir des formalismes décidables.

Inspirés par la question de représentation des langages temporisés, Bouyer, Petit et Thérien se sont intéressés à une classe décidable d'automates à registres. Dans [BPT03], ils proposent une classe de langages de mots de données qui admet une caractérisation en terme d'automates à registres uni-directionnels, une caractérisation algébrique et même logique. Cette classe de langages est décidable mais on peut noter que les formalismes proposés ne permettent pas d'exprimer des propriétés de clés[2] qui font partie des objectifs de cette thèse.

Les travaux les plus proches de cette thèse sont les récents travaux de Demri, Jurdziński et Lazić présentés dans [DL08a, JL07, JL08, Laz08]. Tous ces travaux sont reliés aux modèles alternants des automates à registres. Ces travaux visent à définir des formalismes génériques pour décrire des langages de mots de données.

[2]comme par exemple “Toutes les positions du mot portent une donnée différente.”

Les auteurs définissent des extensions des logiques temporelles et du μ-calcul à l'aide de registres et la classe des automates alternants à registres sur les mots et les arbres de données. Ils explicitent ensuite les liens entre ces différents formalismes et la frontière de décidabilité. Le vide de ces automates n'est pas forcément décidable, les auteurs donnent ainsi la frontière de décidabilité des différentes sous-classes que l'on peut définir en fonction du nombre de registres et des déplacements de la tête de lecture. En terme d'expressivité les formalismes décidables présentés dans ces travaux sont incomparables aux travaux de cette thèse. On reviendra sur ces travaux dans la première partie.

La logique monadique du second ordre étendue aux mots de données est indécidable [NSV04]. On s'intéresse dans cette thèse à la logique du premier ordre étendue aux mots et arbres de données. Le fragment restreint aux formules utilisant deux variables est décidable et correspond dans le cas des mots à une classe d'automates appelée automates de données. Ces automates correspondent dans le cas des mots aux automates présentés par Björklund et Schwentick dans [BS07]. On explicite les résultats ainsi que les liens entre ces formalismes dans la thèse. Une version étendue des automates a été étudiée par Björklund et Bojanczyk dans [BB07a] dans le cas des arbres de profondeur bornée. Cette restriction permet d'autoriser plus de pouvoir navigationnel à l'automate sans basculer dans l'indécidabilité.

Dans [BB07b], Björklund et Bojanczyk s'intéressent à un modèle étendu des mots de données où chaque position est étiquetée par un n-uplet de données. Le fragment décidable de la logique du premier ordre présenté dans cette thèse devient indécidable sur ce type de modèle (voir Proposition 2.104). Afin d'obtenir un fragment logique décidable, les auteurs imposent une restriction sur la structure des données sur l'ensemble du mot.

Tous ces travaux identifient des langages de mots en exprimant des contraintes d'égalité et de différence sur les valeurs de données entre les différentes positions du mot. C'est aussi ce qui nous intéresse dans cette thèse. Si on ne considère que des tests d'égalité et de différence, on peut aussi voir la partie donnée du mot (ou d'un arbre) de données comme une relation d'équivalence. Kieronski et Otto montrent dans [KO05] que la logique du premier ordre à deux variables est indécidable si on autorise au moins trois relations d'équivalence.

Les bases de données semi-structurées

Bien que les données représentent une composante essentielle des documents, les premiers travaux théoriques sur XML se sont concentrés sur la structure finie des documents : arbre étiqueté par un alphabet fini. On a ainsi mis en évidence les nombreux liens entre les formalismes classiques tels que la logique, les automates, les grammaires et les langages XML. Les DTD (Document Type Definition) peuvent être vues comme des grammaires ou des automates sur les arbres d'arité non bornée. Le fragment navigationnel du langage X-Path, Core-XPath, correspond au fragment à deux variables de la logique du premier ordre sur les arbres [GKP02, Mar05]. Core-XPath est décidable même en présence de DTD et la complexité des différents fragments a été étudiée en détails. On peut se référer aux travaux et références présentés par Benedikt et al dans [BFG05, GF05] pour plus de détails.

Lorsque l'on veut tenir compte des données, on se heurte rapidement à des problèmes d'indécidabilité. Par exemple on peut considérer le langage Core-data-XPath correspondant à l'extension de Core-XPath qui autorise simplement des tests d'égalité et de différence entre les valeurs d'attributs. Dans [GF05], les auteurs montrent que Core-Data-XPath est indécidable. L'étude des différents fragments de Core-Data-XPath présentés dans [BFG05, GF05] montre qu'il est tout de même possible d'obtenir des résultats de décidabilité tout en permettant des tests d'égalité sur les données mais cela nécessite d'importantes restrictions sur le langage, en particulier en matière de navigation.

Si on considère les DTDs, une manière naturelle pour tenir compte des données dans la spécification de schémas est de coupler ces DTDs avec des contraintes d'intégrité. Dans [AFL05] Arenas et al s'intéressent au problème de cohérence de ce type de schémas. D'une manière générale, le problème est indécidable mais on obtient la décidabilité si on se restreint à des contraintes de type clés unaire et clés étrangères. On peut aussi remarquer que ces résultats portent sur les DTDs dites couplées où chaque tag est associé à un seul type.

Les résultats de cette thèse se replacent naturellement dans ce cadre. Le fragment à deux variables de la logique du premier ordre étendue aux arbres de données permet d'exprimer des schémas incluant des DTDs couplées à des contraintes sur les données comme par exemple des contraintes de clé unaire et des contraintes de clés étrangère. On peut aussi traduire certains fragments de Core-Data-XPath dans cette logique. On unifie ainsi des formalismes étudiés habituellement séparément. Les résultats d'indécidabilité et décidabilité obtenus dans cette thèse donnent une idée plus précise de la frontière de décidabilité des problèmes d'analyse statique liés aux documents XML. On retrouve entre autres le résultat de décidabilité pour les contraintes unaires obtenu dans [AFL05].

La seconde approche proposée dans cette thèse a une expressivité différente de XPath. On propose un formalisme à base de motifs d'arbres (*patterns*) qui permet d'exprimer des contraintes sur les données. En combinant ces contraintes à des langages réguliers d'arbres on obtient un formalisme de spécification de schéma puissant et orthogonal à la logique précédemment proposée. En particulier les formules de patterns permettent d'exprimer des contraintes de clés (sans se limiter aux clés unaires) mais ne peuvent pas exprimer de contrainte de clé étrangère. Dans ce cas encore, on obtient des résultats d'(in)décidabilité à ajouter aux résultats précédents.

Les travaux de Bidoit, Cerrito, Colazzo et Thion présentés dans [BCT04, BC07] sont aussi motivés par la recherche de formalismes génériques de description de schémas, de contraintes sur les données et de requêtes de type XPath. Les auteurs proposent une extension de logique modale appelée HML qui permet entre autres d'exprimer des contraintes de typage sur les références présentes dans les schémas. La logique HML est indécidable en général, mais décidable si on considère le fragment restreint aux schémas non récursifs. Les auteurs décrivent alors une méthode basée sur les tableaux qui permet d'obtenir la décidabilité de la logique dans le cas général des arbres finis et infinis. Ils donnent des pistes pour obtenir la décidabilité de la logique dans le cas des arbres finis.

Enfin, les problèmes liés aux données apparaissent aussi dans des domaines qui dépassent le cadre de l'analyse statique. On peut par exemple mentionner les travaux d'Abiteboul, Segoufin et Vianu présentés dans [ASV06] qui portent sur des documents actifs c'est-à-dire des documents qui évoluent avec le système.

On se heurte encore une fois à des problèmes d'indécidabilité dans la recherche de formalismes adaptés pour décrire des propriétés de ces systèmes évolutifs.

Spécification et vérification de systèmes

Si on se place maintenant dans le cadre de la spécification et vérification de systèmes, on voit apparaître des structures avec des données non-bornées dans les premiers travaux de Alur et Dill sur les systèmes temporisés [AD90]. On utilise des horloges dans l'abstraction du système afin de représenter le temps. Les modèles des exécutions du système sont alors munis des valeurs des horloges, on parle de mots temporisés. Dans ce cadre, les données considérées sont alors des nombres (réels, rationnels, ou entiers). Selon les cas, l'ensemble des valeurs d'horloge est muni de la relation d'ordre, des tests d'égalité et de comparaison à une constante ou même d'opérations arithmétiques restreintes. Les travaux dans le domaine ont cherché à étendre les formalismes classiques. En matière d'automates, le modèle le plus connu est celui des automates temporisés [AD90, AD94] qui correspond à une extension des automates classiques à l'aide d'horloges qui évoluent de manière continue avec le temps. Les transitions de ces automates sont contraintes par des tests (appelés "gardes") et peuvent effectuer des opérations (comme par exemple la mise à zéro d'une horloge). Au niveau logique, on définit des extensions des logiques temporelles linéaires ou arborescentes en modifiant les opérateurs classiques pour ajouter des contraintes temporelles [Koy90, EMSS92, AD93, AH93]. Il existe de nombreuses sous-classes et extensions de ces différents formalismes. Comme dans le cas classique il existe des liens entre les formalismes logiques et les modèles d'automates [LLW95, Wil94] ; la recherche dans ce domaine a été très active depuis plus d'une dizaine d'années. Par rapport à cette thèse, ce domaine manipule des structures contenant des données non-bornées d'un type particulier (ensemble de nombres munis selon les cas de plus ou moins d'arithmétique). Les travaux de Bouyer et al présentés précédemment [BPT03] ont été inspirés par les langages temporisés. D'une manière générale, il est important de remarquer que les problématiques de la décidabilité et de la complexité des problèmes relatifs à ces langages temporisés fait partie des principales difficultés du domaine.

D'autre types de données ont ensuite été considérées; au lieu de raisonner sur le temps, on s'intéresse par exemple au nombre d'occurrences de certains événements du système. Les travaux présentés dans [BEH95, BH96] considèrent des logiques temporelles permettant, entre autres, d'exprimer des contraintes sur le nombre d'événements observés à l'aide de formules de Presburger. Encore une fois, la décidabilité des fragments considérés est une question clé de ces travaux. Les travaux présentés dans [SSMH04] étudient une extension des automates en ajoutant des contraintes de comptage et montrent que cela ne change pas la décidabilité du problème du vide mais augmente sa complexité. Les récents travaux présentés dans [DL08b] étendent ces résultats en s'intéressant à des extensions de logique modale.

De manière plus générale, les travaux présentés dans [DG08, DLN07] étudient des logiques munies de systèmes de contraintes portant sur des variables prenant leurs valeurs dans des domaines de données variés tels que $(\mathbf{N}, +, =)$. Dans [DLS08] les auteurs s'intéressent à la logique temporelle à registres présentée dans [DL06] dans le cadre des exécutions d'automates à un compteur (l'ensemble des exécutions correspond à un langage de mots de données).

Enfin on retrouve la problématique des données non-bornées dans l'étude de systèmes temporisés distribués. On peut citer dans ce contexte les travaux présentés dans [ADR07] ou encore [BJS07]. Enfin dans [BHJS07], les auteurs s'intéressent au problème de satisfaisabilité ou à la propriété d'invariance à travers les transformations de structures avec données (mots, arbres, multi-ensembles) permettant de représenter des configurations d'un système.

D'une manière générale dans le domaine de la spécification et la vérification de systèmes, les travaux relatifs aux données non-bornées sont actuellement nombreux. Afin de restreindre le champ d'application dans le but d'obtenir des résultats positifs, ces travaux consistent en général à considérer des structures de données particulières afin de voir ce que l'on peut autoriser en restant décidable. On considère ainsi des tableaux, des listes, des mots, des arbres. Selon les cas on va aussi s'intéresser à des données de types variés comme l'ensemble des entiers muni d'opérations arithmétiques ou au contraire un ensemble d'identifiants de processus muni d'un simple test d'égalité. Les résultats s'appuient ainsi fortement sur le type de donnée ainsi que les structures manipulées. Les opérations atomiques autorisées dans les langages de spécifications sont plus nombreuses que le seul test d'égalité mais les formules autorisées sont en contrepartie très restreintes.

Même si la problématique de gestion des données est commune à notre travail, ces travaux sont souvent incomparables soit de par le type de données ou de structures considérées soit aussi de par les propriétés considérées dans les différents cas d'application.

Organisation du manuscrit

Dans une première partie, on présente les travaux relatifs aux formalismes logiques et aux modèles d'automates. Le Chapitre 2 est consacré aux mots de données. On y rappelle les résultats relatifs aux logiques temporelles ainsi qu'aux modèles d'automates à registres. On présente ensuite une extension de la logique du premier ordre ainsi que le modèle des automates à données et les résultats associés obtenus pendant cette thèse. Le Chapitre 3 est consacré au cas des arbres de données. On commence par présenter rapidement le langage XML et les problématiques associées qui permettent de mettre en relief le travail de cette thèse. On explique dans quelle mesure les formalismes et les résultats s'étendent au cas des arbres et on relie ces résultats aux problématiques XML présentées au début du chapitre.

La seconde partie est consacrée à l'étude d'un formalisme basé sur des combinaisons booléennes de "patterns". Le Chapitre 4 est consacré au cas des arbres de données. Après avoir défini ce nouveau formalisme, on s'intéresse successivement au problème du model-checking et de satisfaisabilité. Le Chapitre 5 reprend cette étude dans le cas restreint des mots de données. Le chapitre 6 est un chapitre de discussion contenant de nombreuses questions et remarques relatives au modèle des "patterns" : extensions, expressivité par rapport aux formalismes précédents et liens avec les problématiques XML.

Le Chapitre 7 redonne les principaux résultats et perspectives de cette thèse.

Première partie

Logiques et Automates

Le but de cette partie est de définir des formalismes logiques afin de pouvoir décrire des propriétés sur des modèles contenant des valeurs de données. On s'intéresse dans un premier temps à un modèle étendu des mots finis appelés mots de données. Un mot de données est un mot fini dont chaque position est étiqueté d'une part par une lettre d'un alphabet fini et d'autre part par une donnée provenant d'un domaine quelconque (potentiellement infini). L'objectif est d'étendre les formalismes logiques sur les mots étiquetés par un alphabet fini tels que la logique temporelle ou la logique du premier ordre. Il est naturel dans ce contexte de chercher en parallèle à obtenir des modèles d'automates équivalents. Les automates constituent en général des outils efficaces pour résoudre des problèmes comme l'appartenance ou la satisfaisabilité.

On va voir qu'étendre la logique et les automates afin de pouvoir faire des tests d'(in)égalité entre valeurs de données est loin d'être facile, du moins si cherche à obtenir des formalismes expressifs dont le problème de satisfaisabilité (ou du vide) est décidable. On présente d'une part une extension de la logique temporelle et un modèle d'automate équivalent utilisant des registres afin de gérer les données ; d'autre part, on propose une extension de la logique du premier ordre à l'aide d'un prédicat de test d'(in)égalité de données et un modèle d'automate correspondant. On trace la frontière de décidabilité associée à ces formalismes et on montre que la complexité du problème de décidabilité est très élevée même dans le cas de fragments moins expressifs. Dans un deuxième temps, on montre dans quelle mesure on peut étendre ces résultats au modèle des arbres de données. Comme les mots de données, les arbres de données sont des arbres (ordonnés, d'arité non-bornée) bi-étiquetés par un alphabet fini et un ensemble infini de données. Les arbres de données sont un modèle théorique permettant de représenter des documents XML. On montrera que les résultats sur les arbres permettent de dessiner plus précisément la frontière de décidabilité d'un ensemble de problèmes dans le domaine de l'analyse statique de documents XML.

La plupart des résultats de cette partie proviennent des travaux de Stéphane Demri, Ranko Lazić et Marcin Jurdziński en ce qui concerne la logique temporelle d'une part [DL08a, JL07, JL08] et de ceux de Mikołaj Bojańczyk, Anca Muscholl, Luc Segoufin, Thomas Schwentick et moi-même en ce qui concerne la logique monadique d'autre part [BMS+06, BDM+06].

Organisation de la Partie :

On commence par définir le modèle des mots et arbres de données ainsi que quelques notions associées.

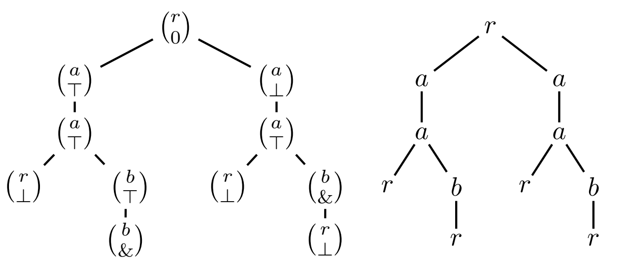
Le Chapitre 2 est consacré aux mots de données. Dans la Section 2.1, on définit la logique temporelle sur les mots de données. On montre que le problème de satisfaisabilité de cette logique est indécidable en général. On présente ensuite différents modèles d'automates à registres en s'intéressant plus particulièrement à la décidabilité du problème du vide pour ces modèles afin de comprendre où se situe la limite de décidabilité et dans quelle mesure les registres augmentent le pouvoir expressif des automates. Enfin, on revient à la logique temporelle en explicitant des fragments dont la décidabilité est reliée aux résultats présentés sur les automates à registres. En matière de complexité, les fragments décidables sont non primitifs récursifs.

Dans la suite du chapitre, on s'intéresse à l'extension de la logique monadique (du premier ordre principalement) à l'aide d'un prédicat de test d'égalité sur les données. Comme dans le cas de la logique temporelle, le fragment général est indécidable. On définit un autre modèle d'automates, appelés automates à données, dont le vide est décidable. A l'aide de ces automates, on montre la décidabilité du problème de satisfaisabilité pour le fragment des formules utilisant au plus deux variables. En matière de complexité, on montre que la satisfaisabilité de ce fragment est aussi difficile que le problème d'accessibilité dans les réseaux de Petri, problème dont on ne connaît pas de borne supérieure élémentaire et donc la meilleure borne inférieure connue est ExpSpace-difficile. On s'intéresse à la complexité de fragments de cette logique. On décrit ensuite plus précisément la frontière de décidabilité en étudiant des extensions naturelles de ce fragment. L'étude de ces extensions permet aussi de donner une caractérisation logique des langages reconnus par les automates à données.

Dans le Chapitre 3, on s'intéresse plus particulièrement aux cas des arbres de données. La première section est une présentation du langage XML et de problématiques associées. En particulier, il est important de pouvoir décrire des ensemble de documents (à l'aide de ce qu'on appelle des schémas) et des ensembles de nœuds dans un document (à l'aide de requêtes) en imposant des contraintes sur les valeurs de données. On montre que les problèmes classiques de cohérence, d'inclusion de schéma, ou encore d'équivalence de requêtes peuvent être vus comme des problèmes de satisfaisabilité. On explique enfin comment coder un document XML dans un arbre de données. Les deux sections suivantes expliquent dans quelle mesure on peut étendre les formalismes du Chapitre 2 au modèle des arbres de données. Le modèle des automates alternants s'étend au cas des arbres et on donne une caractérisation logique de ces automates par un fragment de μ-calcul étendu aux arbres de données. Dans le cas de la logique du premier ordre, on montre la décidabilité d'un fragment à deux variables. Dans la dernière section, on revient au langage XML afin de relier les résultats sur les arbres des deux sections précédentes aux problématiques présentées dans la première section.

$$\begin{array}{ccccccccccccccccccccccccccccc} b & l & t & o & c & l & e & o & m & a & r & m & u & e & e & s & r & n & a & e & u & n & r & g & e & e & e & t & e \\ \alpha_3 & \alpha_0 & \alpha_1 & \alpha_7 & \alpha_4 & \alpha_3 & \alpha_1 & \alpha_4 & \alpha_4 & \alpha_0 & \alpha_7 & \alpha_4 & \alpha_6 & \alpha_3 & \alpha_2 & \alpha_2 & \alpha_1 & \alpha_6 & \alpha_7 & \alpha_4 & \alpha_3 & \alpha_7 & \alpha_1 & \alpha_7 & \alpha_6 & \alpha_1 & \alpha_7 & \alpha_2 & \alpha_3 \end{array}$$

(a) Un mot de données



(b) un arbre de données

(c) L'arbre correspondant à sa projection sur l'alphabet fini

Fig. 1.1 – Exemples

Mots et arbres de données

Dans toute la suite, Σ représente un alphabet fini et $\mathcal{D}$ un ensemble infini de données muni d'une relation d'égalité $\sim$.

Définition 1.1 Un **mot de données** est un mot de $(\Sigma \times \mathcal{D})^*$. Chaque position v est étiquetée par une *lettre* $v.l \in \Sigma$ et une *donnée* $v.d \in \mathcal{D}$.

Définition 1.2 Un **arbre de données** est un arbre (a priori) ordonné d'arité non bornée étiqueté par $\Sigma \times \mathcal{D}$. Plus précisément, chaque nœud v est étiqueté par une *lettre* $v.l \in \Sigma$ et une *donnée* $v.d \in \mathcal{D}$.

De la même manière on définit un mot de données infini comme un mot de longueur ω étiqueté par $\Sigma \times \mathcal{D}$ (ie un mot de $(\Sigma \times \mathcal{D})^\omega$) et un arbre de données infini comme un arbre infini étiqueté par $\Sigma \times \mathcal{D}$. En dehors des quelques cas où l'on fait explicitement référence aux mots ou arbres infinis, on s'intéresse, dans cette thèse, à des structures finies.

La Figure 1.1 de la page précédente donne un exemple de mot et d'arbre de données.

Définition 1.3 Une **classe** d'un mot ou arbre de données w désigne un ensemble maximal de positions étiquetées par une même donnée.

Définition 1.4 La **projection sur l'alphabet fini** d'un mot (resp. arbre) de données w est le mot (resp. arbre) correspondant à la projection du mot (resp. arbre) de données sur l'alphabet Σ. On la note $Str(w)$

Dans la suite on s'intéresse à différents formalismes (logiques et automates) permettant de décrire des langages de mots et arbres de données. Le principal problème que l'on cherche à résoudre est de savoir décider si un langage ainsi décrit est vide. Ce problème dit de *satisfaisabilité* dans le cas de la logique dénommé problème du *vide* dans le cas des automates.

Problème 1.5 Le problème de **Satisfaisabilité** pour une logique $\mathcal{F}$ est le suivant : Etant donné une formule de la logique, existe-t-il un mot/arbre de données qui vérifie la formule ?

Dans le cas des automates, on formule le problème de la manière suivante :

Problème 1.6 Le problème du **Vide** pour une classe d'automates $\mathcal{C}$ est le suivant : Étant donné un automate de la classe $\mathcal{C}$, existe-t-il un mot/arbre de données accepté par l'automate ?

Chapitre 2

Les mots de données

2.1 Registres et Logique temporelle

Dans cette section, on s'intéresse à une approche de gestion des données par l'utilisation de registres afin de stocker des valeurs de données.

On commence par définir la logique temporelle sur les mots de données. On montre que le problème de satisfaisabilité de cette logique est indécidable en général. On présente ensuite différents modèles d'automates à registre en s'intéressant plus particulièrement à la décidabilité du problème du vide pour les différents modèles afin de comprendre où se situe la limite de décidabilité et dans quelle mesure ces registres augmentent le pouvoir expressif des automates. Enfin, on revient à la logique temporelle en explicitant différents fragments dont la décidabilité est reliée aux résultats présentés sur les automates à registres.

Cette partie est descriptive et donne une vue d'ensemble de l'utilisation des registres dans le cas des mots de données. La plupart des résultats sont issus des travaux présentés dans [DLN07, DL06, DL08a, JL07].

2.1.1 L'opérateur freeze - Indécidabilité

Définition 2.1 Étant donné un ensemble d'opérateurs temporels O, l'ensemble des formules de la logique $\mathrm{LTL}^{\downarrow}(O)$ est défini par la grammaire suivante :

$$\phi ::= a \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \mathtt{O}(\phi, \ldots, \phi) \mid \ \downarrow_r \phi \mid \ \uparrow_r$$

où a est une lettre de l'alphabet fini Σ, r est un entier strictement positif désignant un registre et $\mathtt{O} \in O$ un opérateur temporel. Une occurrence de $\uparrow_r$ est dite liée si elle est dans le champ d'action d'un $\downarrow_r$, et libre sinon.

On considère les opérateurs temporels "suivant" ($\mathtt{X}$), "un jour dans le futur" ($\mathtt{F}$), "tant que" ($\mathtt{U}$) et leurs symétriques dans le passé. En présence de l'opérateur $\mathtt{U}$, l'opérateur $\mathtt{F}$ peut être omis car il s'exprime comme $\top\mathtt{U}\phi$. De la même manière, on ne mentionne pas l'opérateur "toujours" ($\mathtt{G}$) qui s'exprime facilement à l'aide de l'opérateur $\mathtt{F}$ et de la négation.

On note $\mathrm{LTL}^{\downarrow}_n(O)$ le fragment des formules qui utilisent au plus n registres ($r \in \{1, \ldots, n\}$). Dans la suite, on ne parlera que des fragments à un ou deux registres.

Sémantique. Les modèles des formules sont les mots de données (finis ou infinis selon les cas). On considère une formule ϕ, un mot de données w, une position i du mot et une fonction μ qui associe à chacun des registres de la formule soit une valeur de données de $\mathcal{D}$ soit le symbole #. On note $w, i \models_\mu \phi$, le fait que la formule ϕ est satisfaite par w à la position i en respectant μ. La sémantique des formules est la sémantique usuelle de LTL à laquelle il faut ajouter la gestion des données à travers le $\downarrow$:

$w, i \models_\mu a$	$\leftrightarrow$	La position i est étiquetée a
$w, i \models_\mu \mathtt{X}\phi$	$\leftrightarrow$	$i < \lvert w \rvert$ et $w, i+1 \models_\mu \phi$
$w, i \models_\mu \mathtt{X}^{-1}\phi$	$\leftrightarrow$	$1 < i$ et $w, i-1 \models_\mu \phi$
$w, i \models_\mu \phi \mathtt{U} \psi$	$\leftrightarrow$	Il existe $j \geq i$ tel que $w, j \models_\mu \psi$ et pour tout k tel que $i \leq k < j$ on a $w, k \models_\mu \phi$
$w, i \models_\mu \phi \mathtt{U}^{-1} \psi$	$\leftrightarrow$	Il existe $j \leq i$ tel que $w, j \models_\mu \psi$ et pour tout k tel que $j < k \leq i$ on a $w, k \models_\mu \phi$
$w, i \models_\mu \downarrow_r \phi$	$\leftrightarrow$	$w, i \models_{\mu_{[r \mapsto i.d]}} \phi$ où $i.d$ est la valeur de données à la position i du mot w.
$w, i \models_\mu \uparrow_r$	$\leftrightarrow$	la position i porte la donnée $\mu(r)$

A l'aide d'un codage des exécutions acceptantes des machines à deux compteurs, on peut montrer le résultat suivant :

Théorème 2.2 ([DL06]) *Le problème de satisfaisabilité pour la logique* $LTL^{\downarrow}(X, U)$ *est indécidable.*

On va donc considérer des fragments de cette logique.

2.1.2 Les automates à registres

Les automates à registres sont des automates finis munis d'un nombre fini de registres. Au cours d'une exécution, l'automate peut ainsi stocker temporairement des valeurs de données pour ensuite les comparer avec la valeur de données de la position courante. Ces automates ont été introduits dans [KF94] puis repris dans [NSV04].

Définition 2.3 Un automate à registres $\mathcal{A}$ est un tuple $(Q, q_0, Q_f, k, \mu_0, \delta)$ où :

- Q est l'ensemble fini des états, q_0 est l'état initial et Q_f l'ensemble des états finaux,
- $k \in \mathbb{N}$ est le nombre de registres,
- $\mu_0 : [1 \ldots k] \to \mathcal{D}$ est la valuation initiale des registres.
- δ est l'ensemble fini des transitions. Une transition de l'automate est de la forme $(q, a, \theta^{\sim}, \theta^{\nsim}) \to (q', r)$ où $q, q' \in Q$, $a \in \Sigma$, $r \in \{\#, 1, \ldots, k\}$ et $\theta^{\sim}, \theta^{\nsim} \subseteq \{1, \ldots, k\}$.

Étant donné un mot de données w, une configuration de l'automate $\mathcal{A}$ sur w est un triplet (i, q, μ) où $0 \leq i < |w|$ est une position du mot, $q \in Q$ et $\mu : \{1, \ldots, k\} \to \mathcal{D}$ est la valuation courante des registres. La configuration initiale est $(0, q_0, \mu_0)$. Une configuration est acceptante si l'état courant est un état final.

Étant donné une configuration (i, q, μ), l'automate peut effectuer la transition $(q, a, \theta^{\sim}, \theta^{\nsim}) \to (q', r)$ si et seulement si la position courante est étiquetée a et sa valeur de données vérifie des contraintes d'(in)égalités données par $\theta^{\sim}$ et $\theta^{\nsim}$: La donnée de la position i doit être égale aux valeurs stockées dans les registres désignés par $\theta^{\sim}$ et différente des valeurs stockées des registres désignés par $\theta^{\nsim}$. Si $r = \#$, la nouvelle configuration est $(i+1, q', \mu)$. Sinon, la nouvelle configuration est $(i+1, q', \mu')$ telle que μ' est identique à μ sauf pour le registre r qui contient alors la valeur de données de la position i.

Proposition 2.4 [KF94] La classe des automates à registres n'est pas close par complément et les automates non-déterministes sont plus puissants que les automates déterministes.

Exemple 2.5 Au moins deux positions dans la même classe. On peut reconnaître avec un automate à registres le langage $L_{\sim}$ correspondant à l'ensemble des mots de données qui contiennent deux positions dans une même classe. L'automate (non déterministe) utilise un seul registre. Il devine une première position dont il stocke la valeur de données dans le registre puis vérifie que cette donnée apparaît bien à nouveau dans le mot en comparant successivement les valeurs de données suivantes avec la valeur stockée.

On peut montrer que le langage $L_{\sim}$ ne peut pas être reconnu par un automate à registres déterministe.

Le complément de ce langage, $L_{\nsim}$, est l'ensemble des mots dont toutes les positions sont dans des classes différentes. $L_{\nsim}$ n'est pas reconnu par un automate à registres. En effet, on peut montrer que pour tout mot de données accepté par un automate à registres, il existe un mot reconnu par le même automate qui a la même projection sur l'alphabet fini mais un nombre de classes borné par une constante dépendant uniquement de l'automate (voir Proposition 2.9).

De cette remarque on en déduit :

Théorème 2.6 ([KF94]) *Le vide des automates à registres est décidable.*

En matière de complexité le problème du vide est PSPACE-complet [DL06] et dans NLOGSPACE si on fixe le nombre de registres. Si l'alphabet fini contient une seule lettre (ce qui revient à considérer des mots de $\mathcal{D}^*$) le problème est NP-complet [SI00]. Malheureusement, ce modèle d'automate n'est pas robuste.

Théorème 2.7 ([NSV04]) *Le problème d'universalité des automates à registres est indécidable.*

Du Théorème précédent, on en déduit aussi :

Proposition 2.8 ([NSV04]) Les problèmes de l'inclusion et d'équivalence pour les automates à registres sont indécidables.

En matière d'expressivité, on a expliqué que les automates à registres ne peuvent pas décrire le langage $L_{\nsim}$. Ce modèle d'automate ne peut donc pas exprimer les propriétés de clés qui nous intéressent. D'autre part, on sait caractériser la projection finie des langages reconnus par ces automates.

Proposition 2.9 ([BPT03, KF94]) Étant donné un automate à registres, l'ensemble des projections sur l'alphabet fini des mots reconnus par cet automate constitue un langage régulier.

Une manière naturelle d'augmenter le pouvoir expressif de ces automates est de considérer des modèles bidirectionnels ou alternants. Malheureusement, on perd alors la décidabilité.

Théorème 2.10 ([Dav04]) *Le vide des automates bidirectionnels à un registre est indécidable.*

Dans [NSV04], les auteurs prouvent ce résultat sans restrictions sur le nombre de registres. Le résultat montré dans [Dav04] est en réalité plus puissant que le Théorème 2.10 car il concerne une classe restreinte des automates bidirectionnels appelée automates à un registre partiellement ordonnés déterministes.

Dans la suite on s'intéresse plus en détails à la notion d'automates alternants et aux liens entre ces automates et la logique temporelle sur les mots de données.

2.1.3 Automates alternants et logique temporelle

Le modèle des automates à registres alternants est étudié par Stéphane Demri et Ranko Lazić dans les articles suivants : [DL06, DL08a]. La différence avec les automates définis ci-dessus est la définition de la relation de transition qui permet des branchements alternant. Ces travaux explorent en détails des questions de d'expressivité, de décidabilité et de complexité de ce modèle d'automate dans le cas des mots finis ainsi que dans celui des mots infinis. On présente ici les résultats relatifs aux mots finis de données qui font l'objet de cette thèse. Le cas des mots infinis est rapidement mentionné dans la section 2.1.4. Le lecteur est invité à consulter les articles ci-dessus pour plus de détails sur le sujet.

Définition 2.11 Un automate alternant à registres est un automate faible à registres tel que : étant donnée une position dans un mot de données, une valuation des registres et un état de l'automate, une transition de l'automate permet d'effectuer une des opérations suivantes :
- changer d'état en fonction de la position courante : de son étiquette, de sa valeur de données comparée à la valuation des registres et le fait que position courante soit la première ou la dernière du mot ;
- stocker la valeur de données courante dans un registre ;
- effectuer un branchement (disjonction ou conjonction) vers deux états de l'automate. On parle de branchement existentiel ou universel ;
- accepter ou rejeter le mot ;
- déplacer la tête de lecture dans la position suivante ou précédente du mot.

On note ARA_k la classe des automates alternants à k registres.

Théorème 2.12 ([DL06]) *La classe ARA_k est close par complément. La construction est effective et se fait en espace logarithmique dans la taille de l'automate.*

Le vide des automates à registres unidirectionnels non alternants est décidable mais si on considère le modèle alternant le problème du vide est indécidable si on autorise au moins deux registres [DL06]. En restreignant le modèle à l'utilisation d'un seul registre, on garde la propriété de décidabilité du vide dans le cas des automates alternants.

Théorème 2.13 ([DL06]) *Le problème du vide des automates ARA_1 est décidable non primitif récursif.*

L'idée de la preuve est de montrer que pour tout automate de type ARA_1, il existe un automate à compteurs à erreurs d'incrémentation qui reconnaît la projection sur l'alphabet fini du langage. La classe des automates à compteurs à erreurs d'incrémentation correspond à des automates munis de compteurs. Les transitions peuvent incrémenter ou décrémenter les compteurs et faire des tests à zéro. De plus, les transitions contiennent des erreurs d'incrémentation : à tout moment un compteur peut-être incrémenté d'une valeur positive quelconque. Le vide de ces automates est décidable sur les mots finis mais la complexité du problème est non-primitive récursive.

Théorème 2.14 ([DL06]) *Pour toute formule de la logique $LTL^{\downarrow}_n(\mathtt{X}, \mathtt{U})$, on peut construire un automate de la classe ARA_n qui reconnaît le même langage. Cette construction peut se faire en espace logarithmique dans la taille de le formule.*

Corollaire 2.15 Le problème de satisfaisabilité pour $\mathrm{LTL}^{\downarrow}_1(\mathtt{X}, \mathtt{U})$ est décidable.

On peut obtenir de la même manière que dans le cas des automates la borne inférieure non primitive recursive en codant les executions des automates à compteurs avec erreurs d'incrémentation.

Comme pour les automates, la logique $\mathrm{LTL}^{\downarrow}_1(\mathtt{X}, \mathtt{U})$ avec des opérateurs passés ou en autorisant plus de compteurs, on obtient rapidement l'indécidabilité des problèmes de satisfaisabilité.

Théorème 2.16 ([DL06]) *Les problèmes de satisfaisabilité pour $LTL^{\downarrow}_1(\mathtt{X}, \mathtt{F}, \mathtt{F}^{-1})$, $LTL^{\downarrow}_2(\mathtt{X}, \mathtt{F})$ sont indécidables.*

2.1.4 Les mots infinis

Dans le cas des mots infinis, on peut étendre la définition des automates ARA_1. On définit une notion de rang sur les états et on considère des automates faibles. Les transitions de ces automates à partir d'un état donné se font vers des états de rang inférieur ou égal. Au cours d'une exécution d'un automate faible, le rang des états visités se stabilise à partir d'un certain point. La définition d'acceptance pour une exécution infinie est donnée par une condition de parité sur la valeur du rang stabilisé.

Le vide de la classe ARA_1 est indécidable dans ce cas mais on obtient la décidabilité du problème si on impose une restriction syntaxique, appelée "safety", sur le rang des états des automates [Laz08].

De la même manière le problème de satisfaisabilité pour la logique $\mathrm{LTL}^{\downarrow}_1(\mathtt{X}, \mathtt{U})$ est indécidable si on autorise des modèles finis et infinis. Afin d'obtenir un fragment décidable, on peut rajouter une restriction syntaxique sur les formules. On parle de "safety $\mathrm{LTL}^{\downarrow}_1(\sim, \mathtt{X}, \mathtt{U})$" pour désigner l'ensemble des formules qui ne contiennent pas d'occurrence de `F` ou `U` sous un nombre impair de négations. Le problème de satisfaisabilité de cette logique est alors ExpSpace-complet sur les mots infinis [Laz08].

2.1.5 Conclusion

Une manière naturelle d'étendre les automates et la logique temporelle afin de tenir compte des valeurs de données est d'utiliser des registres. Le seul fragment décidable est le fragment $\mathrm{LTL}_1^{\downarrow}(\mathtt{X},\mathtt{U})$ à un seul registre et utilisant uniquement les opérateurs temporels futurs. Ce fragment correspond à la classe ARA_1 des automates undirectionnels alternants à un registre. Dans les deux cas la complexité obtenue est très importante puisque le problème de satisfaisabilité (resp. du vide) pour $\mathrm{LTL}_1^{\downarrow}(\mathtt{X},\mathtt{U})$ (resp. ARA_1) est décidable mais non primitif récursif. Si on considère des mots infinis de données, il faut ajouter une restriction syntaxique aux formules ainsi qu'une restriction supplémentaire sur les automates afin d'obtenir un résultat de décidabilité, la complexité obtenue est ExpSpace.

Enfin, on remarque que les fragments décidables ne semblent pas pouvoir d'exprimer des propriétés du type : "Pour toute position a, il existe une position b dans la même classe". Ce type de propriété est un des objectifs du travail de cette thèse.

2.2 Logique du premier ordre : définitions et exemples

Dans la suite du chapitre, on s'intéresse à la logique du premier ordre étendue aux mots de données. L'idée est de rajouter à la logique un prédicat binaire qui permet de tester l'égalité des valeurs de données entre deux positions.

On rappelle que dans le cas classique des mots finis étiquetés par un alphabet fini, la logique du premier ordre est décidable avec une complexité non élémentaire [Sto74]. Dans le cas des mots de données, on montre que le problème de satisfaisabilité est indécidable en général. On présente un modèle d'automates appelés *automates à données* qui permet de définir des langages de mots de données. Un automate à données est la combinaison d'un transducteur et d'un automate fini. Le problème du vide est décidable pour cette classe d' automates. Si on restreint la logique au fragment FO^2 des formules utilisant au plus deux variables, on montre que la satisfaisabilité des formules se réduit au vide des automates à données. Plus précisement, pour toute formule, on peut construire un automate à données qui reconnaît le même langage. En termes de complexité, le problème du vide des automates à données se réduit en temps polynomial au problème d'accessibilité dans les réseaux de Petri qui se réduit lui-même en temps polynomial au au problème de satisfaisabilité de la logique. Le reste du chapitre est consacré à une étude détaillée de différents fragments et extensions de la logique FO^2 et des complexités correspondantes. On donne en particulier une caractérisation logique des langages reconnus par les automates à données. On montre que l'on peut étendre la logique FO^2 au cas des mots infinis sans perdre la décidabilité du problème de satisfaisabilité. Considérer des fragments de la logique permet de diminuer significativement la complexité de la satisfasabilité. Par contre, étendre la logique mène très rapidement à l'indécidabilité.

La plupart des résultats de la fin de ce chapitre correspondent aux travaux présentés dans [Dav04, BMS$^+$06].

2.2.1 Définitions

Définition 2.17 Syntaxe. La **logique du premier ordre** sur les mots de données (notée FO($<, +1, \sim$)) est l'ensemble des formules construites à partir des formules atomiques : $x = y$, $x < y$, $x = y + 1$, $a(x)$ pour $a \in \Sigma$ et $x \sim y$ (où x et y sont des variables) en utilisant la grammaire suivante :

$$\phi := \phi \vee \phi \mid \neg\phi \mid \exists x\phi$$

Sémantique. Le domaine d'interprétation des variables est l'ensemble des positions d'un mot de données. Étant donné un modèle w et une fonction μ qui associe à chaque variable une position du modèle, la sémantique des formules est définie de manière suivante :

$w, \mu \models$	$a(x)$	si	$\mu(x)$ est étiqueté par a dans w
$w, \mu \models$	$x = y$	si	$\mu(x)$ et $\mu(y)$ correspondent à la même position de w
$w, \mu \models$	$y = x + 1$	si	$\mu(y)$ est le successeur de $\mu(x)$ dans w
$w, \mu \models$	$x < y$	si	$\mu(x)$ précède strictement $\mu(y)$ dans w
$w, \mu \models$	$x \sim y$	si	$\mu(x)$ et $\mu(y)$ sont dans la même classe dans w
$w, \mu \models$	$\varphi_1 \vee \varphi_2$	si	$w, \mu \models \varphi_1$ ou $w, \mu \models \varphi_2$
$w, \mu \models$	$\neg\varphi$	si	$w, \mu \not\models \varphi$
$w, \mu \models$	$\exists x\ \psi(x)$	si	il existe μ' telle que $\mu'(y) = \mu(y)$ pour tout $y \neq x$ et $w, \mu' \models \psi(x)$

Remarque 2.18 Si on s'autorise au moins trois variables, on peut exprimer la relation successeur entre deux positions à partir de l'ordre à l'aide de la formule suivante :

$$y = x + 1 \ \leftrightarrow\ x < y \ \wedge\ \forall z\ [\ x < z\ \Rightarrow\ [z = y\ \vee\ y < z]\ ]$$

Dans la suite on s'intéresse aux formules restreintes à deux variables ce qui justifie l'utilisation de la formule atomique $x = y + 1$.

Définition 2.19 La **logique existentielle monadique du second ordre** (notée EMSO) se définit à partir de la logique FO en ajoutant des quantifications existentielles du second ordre à la tête d'une formule FO. Plus précisément EMSO correspond à l'ensemble des formules de la forme :

$$\exists X_0 \dots \exists X_k\ \phi(X_0, \dots, X_k)$$

où les X_i sont des variables monadiques du second ordre et $\phi(X_0, \dots, X_k)$ est une formule de la logique FO augmentée des formules atomiques[1] $x \in X_i$.

Étant donné un modèle w de la formule, les variables monadiques du second ordre sont interprétées comme des ensembles de positions du modèle. La sémantique de la formule atomique correspondante est définie de manière usuelle.

$$w, \mu \models x \in X \text{ ssi } \mu(x) \in \mu(X)$$

Fragments. Dans la suite, on est amené à s'intéresser à des fragments des logiques que l'on vient de définir. On note FO^k la restriction de FO aux formules utilisant au plus k variables. On désigne par FO($<, +1$) l'ensemble des formules de la logique FO($\sim, <, +1$) n'utilisant pas le prédicat binaire $\sim$ d'égalité

[1]Dans la suite, on utilisera parfois la notation $X_i(x)$ à la place de $x \in X_i$.

de données. De la même manière la logique FO($\sim, <$) (resp. FO($\sim, +1$)) correspond aux formules obtenues sans le prédicat $+1$ (resp.$<$). On note EMSO^k, la restriction de EMSO correspondant à l'ensemble des formules de la forme $\exists X_0 \ldots \exists X_n$
$\phi(X_0, \ldots, X_n)$ où la formule $\phi(X_0, \ldots, X_n)$ est une formule de la logique FO^k. De la même manière, on note EMSO($\sim, +1$), EMSO($+1$) ... les autres restrictions de EMSO.

2.2.2 Exemples et formules de base

On présente ici des exemples de formules de la logique.

Remarque 2.20 On remarquera que toutes les formules décrites dans ces exemples sont des formules du fragment FO$^2(\sim, <, +1)$.

Dans la suite, certaines formules sont construites par induction. Il faut faire attention à la réutilisation de ces variables dans les définitions. On se permet le raccourci syntaxique suivant. Si par exemple la formule $\phi(x)$ est définie comme $\forall y\ \psi(x, y)$, on utilise $\phi(y)$ comme étant la formule $\forall x\ \psi(y, x)$ où $\psi(y, x)$ est définie inductivement de la même façon à partir de $\psi(x, y)$.

Exemple 2.21 : D'un boût à l'autre. La formule $debut(x)$ (resp. $fin(x)$) exprime le fait que la position x est la première (resp. la dernière) position du mot. La formule $fin_k(x)$, k étant un entier positif, exprime le fait que la position $x + k$ est la dernière du mot. De la même manière, on définit $debut_k(x)$ qui exprime le fait que la position $x - k$ est la première du mot.

On présente ci-dessous deux façons de définir la formule *debut* en utilisant le prédicat $<$ ou $+1$.

$$debut(x) ::= \forall y\ [x \leq y] \quad \text{ou} \quad \forall y\ [x \neq y + 1]$$

La formule $debut_0(x)$ est la formule $debut(x)$. La formule $debut_k$ se définit par induction comme suit.

$$debut_{k+1}(x) ::= \exists y\ [x = y + 1]\ \wedge\ debut_k(y)$$

On donne l'exemple de cette formule pour $k = 2$.

$$debut_2(x) ::= \exists y\ (x = y + 1\ \wedge\ \exists x\ (y = x + 1\ \wedge\ \exists y\ (x = y + 1\ \wedge\ debut(y))))$$

Exemple 2.22 : La première position vérifiant $\alpha(x)$. Étant donnée une formule α, on note Fst_α (resp. Lst_α) la première (resp. dernière) position du mot qui vérifie α. Par exemple, si α est la formule $a(x)$, Fst_α correspond à la première position étiquetée par un a dans le mot. On note Ffst_α (resp. Llst_α) la première(resp. dernière) position du mot qui vérifie α et qui est dans une classe différente de Fst_α (resp. Lst_α).

$$\mathrm{Fst}_\alpha(x) ::= \alpha(x) \wedge \forall y\ [x \leq y\ \vee\ \neg\alpha(y)]$$

$$\mathrm{Ffst}_\alpha(x) ::= \alpha(x) \wedge \exists y\ [\mathrm{Fst}_\alpha(y) \wedge x \nsim y] \wedge \forall y\ [[y < x\ \wedge\ \alpha(y)] \rightarrow [\exists x\ \mathrm{Fst}_\alpha(x) \wedge x \sim y]]$$

Exemple 2.23 : Compter les a et les b. La formule φ ci-dessous reconnaît un ensemble de mots de données dont la projection sur l'alphabet fini est l'ensemble de tous les mots de $\{a, b\}^*$ qui contiennent autant de positions étiquetées par a que par b.

- La formule φ_a assure que tous les a sont dans des classes différentes.

$$\varphi_a = \forall x \forall y (x \neq y \ \wedge a(x) \wedge a(y)) \rightarrow x \not\sim y$$

  On définit de la même manière φ_b.
- La formule $\psi_{a,b}$ assure que chaque classe qui contient un a contient aussi un b :

$$\psi_{a,b} = \forall x \exists y \big(a(x) \rightarrow (b(y) \wedge x \sim y)\big)$$

  On définit de la même manière $\psi_{b,a}$.
- Un mot de données qui vérifie $\varphi = \varphi_a \wedge \varphi_b \wedge \psi_{a,b} \wedge \psi_{b,a}$ a autant de positions étiquetées par a que de positions étiquetées par b.

On peut facilement étendre ce genre de formule afin de décrire des mots de données ayant le même nombre de a, de b et de c. On obtient ainsi un langage dont la projection sur l'alphabet fini n'est pas hors-contexte.

Exemple 2.24 : Compter jusqu'à n. En réutilisant les variables, on peut construire une formule $\phi_n(x)$ de FO^2, qui vérifie que la classe de x contient au moins n éléments à droite de cette position (incluse). La formule $\phi_1(x)$ est uniformément vraie. La formule $\phi_{n+1}(x)$ se définit de la manière suivante :

$$\exists y \ \ x < y \ \wedge x \sim y \ \wedge \ \phi_n(y)$$

En utilisant cette formule, on peut tester si la classe de x contient au moins n éléments à l'aide de la formule :

$$\exists y \ y \sim x \ \wedge \phi_n(y)$$

Exemple 2.25 : Trouver un mot. La formule $u(x)$ où $u \in \Sigma^*$ est définie par induction de la manière suivante : $a(x) := a(x)$ et $a.u(x) := a(x) \wedge \exists y \ (y = x+1 \ \wedge \ u(y))$. Étant donné un mot de données w, la formule $u(x)$ est vérifiée à la position x si et seulement si la projection sur l'alphabet fini du mot w prise à partir de x commence par le mot u.

2.3 Indécidabilité de la logique du premier ordre avec données

2.3.1 Le problème de correspondance de Post

Toutes les preuves d'indécidabilité que l'on va présenter dans la suite utilisent une réduction du problème de correspondance de Post (PCP).

Problème 2.26 Une instance du **problème de correspondance de Post** (PCP) est donnée par un ensemble fini de paires de mots de Γ^* : $I = (u_i, v_i)_{i \in [1..k]}$. Une solution du problème pour l'instance I est une séquence non-vide d'indices $j_1, \ldots, j_n$ telle que les mots $u_{j_1} u_{j_2} \ldots u_{j_n}$ et $v_{j_1} v_{j_2} \ldots v_{j_n}$ sont égaux.

La question est de savoir, étant donnée une instance du problème, s'il existe une solution.

Théorème 2.27 ([HU79]) *Le problème de correspondance de Post est indécidable.*

De nombreuses variantes et restrictions de ce problème ont été étudiées dans la littérature. Par exemple le problème devient décidable si l'alphabet Γ est composé d'une seule lettre, si le nombre de paires de mots données est inférieur ou égal à deux, si on borne la taille de la séquence solution par un entier, ou encore si on impose que chaque mot u_i (resp. v_i) commence par un symbole différent.

Cependant, certaines restrictions du problème restent indécidables. Dans la Section 2.6, on utilisera la variante suivante :

Théorème 2.28 ([HU79]) *Le Problème de correspondance de Post restreint à des paires de mots de longueur un ou deux est indécidable.*

2.3.2 Indécidabilité de FO^3

Dans cette partie, on montre que le problème de satisfaisabilité pour la logique $\mathrm{FO}(\sim, <, +1)$ est indécidable même si on se restreint aux formules utilisant au plus trois variables.

Théorème 2.29 *Le problème de satisfaisabilité pour la logique* $\mathrm{FO}^3(\sim, <)$ *est indécidable dans le cas des mots de données.*

Preuve. On montre l'indécidabilité du problème de satisfaisabilité pour le fragment $\mathrm{FO}^3(\sim, <, +1)$. D'après la Remarque 2.18, cette logique est égale au fragment $\mathrm{FO}^3(\sim, <)$.

On réduit le problème de correspondance de Post (PCP) au problème de satisfaisabilité des formules de $\mathrm{FO}^3(\sim, <, +1)$.

Étant donnée une instance de PCP, on donne un codage des solutions du problème. On construit alors une formule $\varphi_I \in \mathrm{FO}^3(\sim, <, +1)$ qui reconnaît exactement les codages des solutions de l'instance I.

On considère $I = (u_i, v_i)_{i \in [1..k]}$ une instance de PCP et $j_1, \ldots, j_n$ une solution. On choisit un alphabet $\Sigma = \Gamma \cup \bar{\Gamma}$ où $\bar{\Gamma}$ est une copie de l'alphabet Γ. (Si a est une lettre de Γ, on note $\bar{a}$ sa copie dans $\bar{\Gamma}$.) On code une solution dans un mot de données qui vérifie les propriété suivantes :

(1) La projection sur l'alphabet fini du mot est $\#u_{j_1}\overline{v_{j_1}}\#u_{j_2}\overline{v_{j_2}}\#\ldots\#u_{j_n}\overline{v_{j_n}}\#$ où $\bar{v}_i$ représente le mot v_i écrit avec l'alphabet $\bar{\Gamma}$.

(2) Les positions étiquetées par Γ sont dans des classes différentes.

(3) La séquence de données sur la partie Γ est exactement la même que sur la partie $\bar{\Gamma}$.

Intuitivement chaque position du mot $u_{j_1}u_{j_2}\ldots u_{j_n} = v_{j_1}v_{j_2}\ldots v_{j_n}$ correspond à une valeur de données différente. La Figure 2.1 donne un exemple.

$I = \{(aaa, a), (b, a), (a, ababa)\}$ $\qquad$ $u_1u_2u_3u_2u_3 = v_1v_2v_3v_2v_3 = a\ a\ a\ b\ a\ b\ a\ a\ a$

(a) L'instance du problème $\qquad$ (b) Une solution

$$\begin{array}{cccccccccccccccccccccccc}
\# & a & a & a & \overline{a} & \# & b & \overline{a} & \# & a & \overline{a} & \overline{b} & \overline{a} & \overline{b} & \overline{a} & \# & b & \overline{a} & \# & a & a & a & \overline{a} & \# \\
\perp & d_1 & d_2 & d_3 & d_1 & \perp & d_4 & d_2 & \perp & d_5 & d_3 & d_4 & d_5 & d_6 & d_7 & \perp & d_6 & d_8 & \perp & d_7 & d_8 & d_9 & d_9 & \perp
\end{array}$$

(c) Un codage de cette solution

FIG. 2.1 – Exemple du codage de PCP

Un mot de données w est le codage d'une solution si et seulement s'il vérifie les propriétés suivantes :

(a) $Str(w)$ appartient au langage $\#\{u_i\bar{v}_i\#\}^*{}_{i\in[1..k]}$

(b) Une classe qui contient un $a \in \Gamma$ contient exactement cette position et une position étiquetée $\bar{a}$. Réciproquement une classe qui contient un $\overline{a} \in \bar{\Gamma}$ contient exactement cette position et une position étiquetée a.

(c) Toute paire de données (d_1, d_2) qui apparaît dans le mot sur des positions étiquetées par Γ apparaît dans des positions étiquetées par $\bar{\Gamma}$ dans le même ordre.

La formule $\varphi_{(a)}$ assure la propriété (a). Pour cela, on réutilise la formule définie dans l'Exemple 2.25 ("Trouver un mot"). On note $\varphi_{\#u\bar{v}\#}(x)$ la formule $\bigvee_{i\in[1..k]} \#u_i\bar{v}_i\#(x)$.

$$\underbrace{\exists x\ (debut(x)\ \wedge\ \#(x)\ )}_{(1)} \wedge \underbrace{\forall x\ (\#(x)\ \longrightarrow\ (\varphi_{\#u\bar{v}\#}(x)\ \vee\ fin(x)))}_{(2)}$$

FIG. 2.2 – La formule $\varphi_{(a)}$

La formule $\varphi_{(a)}$ vérifie (1) que la première position du mot est étiquetée par $\#$, (2) que toute position $\#$ est suivie d'un des mots $u_i\bar{v}_i\#$ correspondant à l'instance I sauf s'il s'agit de la dernière position du mot.

Pour exprimer la propriété (b), on procède comme dans l'Exemple 2.23 ("Compter les a et les b"). La conjonction $\varphi_{(b)}$ des deux formules suivantes assure la propriété (b).

$$\bigwedge_{a\in\Gamma} \forall x\ \ a(x) \longrightarrow \Big[\ [\exists y\ (\bar{a}(y) \wedge y \sim x)]\ \ \wedge\ \ [\forall y\ (y \nsim x\ \vee\ y = x\ \vee\ \bar{a}(y))\ ]\Big]$$

$$\bigwedge_{\overline{a}\in\overline{\Gamma}} \forall x\ \overline{a}(x) \longrightarrow \Big[\ [\exists y\ (a(y) \wedge y \sim x)]\ \ \wedge\ \ [\forall y(y \nsim x\ \vee\ y = x\ \vee\ a(y))]\Big]$$

Pour toute position étiquetée par une lettre a de Γ, la première formule vérifie d'une part qu'il existe une autre position dans sa classe étiquetée par la lettre $\bar{a}$ et d'autre part que toute autre position de la classe est de ce type. Cela assure donc en particulier qu'il n'y a qu'une position étiquetée par Γ dans chaque classe. Réciproquement, pour toute position étiquetée par une lettre $\bar{a}$ de $\bar{\Gamma}$, la seconde formule vérifie d'une part qu'il existe une autre position dans sa classe étiquetée par la lettre a et d'autre part que toute autre position de la classe est de ce type.

La propriété (c) est assurée par la formule suivante $\varphi_{(c)}$ où la formule $\Gamma(x)$ correspond à la formule $\bigvee_{a\in\Gamma} a(x)$ et la formule $\overline{\Gamma}(x)$ représente la formule $\bigvee_{\bar{a}\in\overline{\Gamma}} \bar{a}(x)$.

$$\forall x\ \forall y\ [\ \Gamma(x) \wedge \Gamma(y) \wedge x < y\ ] \longrightarrow \exists z\ [\ \overline{\Gamma}(z) \wedge z \sim x \wedge \exists x\ [\overline{\Gamma}(x) \wedge x \sim y \wedge z < y]]$$

La formule φ_I est la conjonction des formules $\varphi_{(a)}$, $\varphi_{(b)}$ et $\varphi_{(c)}$. Par construction, les modèles de la formule sont exactement les codages des solutions de l'instance I du problème PCP. On peut donc en conclure que le problème de satisfaisabilité des formules $\mathrm{FO}^3(\sim, <, +1)$ est indécidable. $\square$

2.4 Les automates à données

Dans cette partie on définit une nouvelle classe d'automates sur les mots de données, appelés *automates à données*. Le vide de ces automates est décidable et on montre que l'on peut réduire le problème de satisfaisabilité des formules FO^2 au vide de ces automates. On obtient ainsi un fragment décidable de la logique FO.

2.4.1 Définitions et exemples

Définition 2.30 Un **automate à données** $(\mathcal{B},\mathcal{C})$ est constitué :
- d'un transducteur lettre à lettre non déterministe $\mathcal{B}$ (appelé l'**automate de base**). L'alphabet d'entrée de ce transducteur est Σ et son alphabet de sortie est Γ ;
- d'un automate fini non déterministe $\mathcal{C}$ (appelé **automate de classe**) qui reconnaît des mots de Γ^*.

Un mot de données $w \in (\Sigma\times\mathcal{D})^*$ est accepté par l'automate à données $(\mathcal{B},\mathcal{C})$ si et seulement s'il existe une exécution acceptante de $\mathcal{B}$ sur $Str(w)$ qui écrit en sortie le mot $b_1 \dots b_{|w|} \in \Gamma^*$, tel que pour chaque classe du mot w représentée par les indices de positions $i_1 < \dots < i_k$, l'automate de classe $\mathcal{C}$ accepte le mot $b_{i_1} \dots b_{i_k}$.

Remarque 2.31 Le fait que le transducteur $\mathcal{B}$ soit lettre à lettre est important pour que le mot $b_{i_1} \dots b_{i_k}$ correspondant à la réécriture d'une classe soit bien défini. Cependant, on peut définir de manière plus générale les automates à données avec des transducteurs quelconques. La définition d'acceptance doit alors être définie avec précaution : l'automate de classe doit accepter les mots correspondant à la réécriture d'une classe du mot de départ par le transducteur. La réécriture d'une classe ne correspond plus forcément à la suite des indices correspondant à la classe dans le mot de départ. Cette définition plus générale ne change pas le pouvoir expressif des automates.

Exemple 2.32 On définit $L_{\not<}$ comme le langage des mots de données w qui vérifient les propriétés suivantes :

1. $Str(w) \in a^*\$\, a^*$
2. La position \$ est seule dans sa classe et les autres classes contiennent exactement deux positions, une située avant le \$ et une située après le \$.
3. L'ordre des valeurs de données dans la partie avant le \$ est différent de celui des valeurs dans la partie suivant le \$.

Le langage $L_{\not<}$ est reconnu par un automate à données. Les deux premières propriétés sont faciles à vérifier à l'aide d'un automate à données (dont l'automate de base est déterministe). On peut vérifier la propriété (3) de la manière suivante : l'automate de base devine et marque deux positions $x < y$ avant la positions \$ et deux positions $x' < y'$ après cette position. Il marque x et y' avec 1 et x' et y avec 0. Ensuite, l'automate de classe accepte les classes qui ne sont pas marquées, ou qui sont formées de deux positions marquées avec la même lettre (0 ou 1).

Si tout automate de classe est équivalent (en termes d'expressivité) à un automate déterministe, la propriété de non-déterminisme du transducteur est essentielle au pouvoir expressif des automates à données.
On dit qu'un automate à données ($\mathcal{B}$,$\mathcal{C}$) est **déterministe** si le transducteur $\mathcal{B}$ et l'automate $\mathcal{C}$ sont déterministes.

Lemme 2.33 Les automates à données détermistes sont strictement moins expressifs que les automates à données.

Remarque 2.34 Sans perte de généralité, on peut supposer que les automates de bases sont complets ; c'est-à-dire qu'il existe une exécution (acceptante ou non) de l'automate pour tout mot de Σ^*.

En effet étant donné un automate à données $\mathcal{A} = (\mathcal{B}, \mathcal{C})$, on peut transformer l'automate de base $\mathcal{B}$ en un automate complet $\mathcal{B}'$ tel que le langage reconnu par l'automate à données $(\mathcal{B}', \mathcal{C})$ est le langage reconnu par $\mathcal{A}$. On procède de la manière suivante. On enrichit l'alphabet de sortie Γ d'une nouvelle lettre $\gamma_\perp$ et on ajoute un nouvel état puits (non-final) $q_\perp$ à l'automate $\mathcal{B}$. Il suffit alors de rajouter les transitions manquantes : si un état q n'a pas de transition pour la lettre $a \in \Sigma$, on rajoute une transition $(q, a, \gamma_\perp, q_\perp)$ à l'automate (de l'état q en lisant a on passe dans l'état $q_\perp$ en écrivant la lettre $\gamma_\perp$). On obtient ainsi un automate $\mathcal{B}'$ complet tel que pour tout mot $w \in \Sigma$ les exécutions acceptantes de l'automate $\mathcal{B}'$ sont exactement les exécutions de l'automate $\mathcal{B}$. En particulier on obtient les mêmes mots de sortie dans Γ^*. Cela n'affecte donc pas les exécutions de l'automate de classe $\mathcal{C}$. Les deux automates $(\mathcal{B}', \mathcal{C})$ et $(\mathcal{B}, \mathcal{C})$ reconnaissent bien le même langage.
Cette transformation préserve la propriété de déterminisme.

Preuve. (Lemme 2.33) L'Exemple 2.32 décrit un langage reconnu par un automate à données. On montre que ce langage n'est reconnu par aucun automate à données dont l'automate de base est déterministe. Plus précisément, on montre que pour tout automate à données déterministe, il existe deux mots de données w et v indistingables par l'automate tels que $w \notin L_{\not<}$ et $v \in L_{\not<}$.

On considère un automate à données déterministe $\mathcal{A} = (\mathcal{B}, \mathcal{C})$. On suppose que $\mathcal{B}$ est déterministe complet et on note n son nombre d'états. On considère un mot de données $w = (a, d_1)(a, d_2)\dots(a, d_{n+1})(\$, d_0)(a, d_1)(a, d_2)\dots(a, d_{n+1})$ dont les valeurs de données d_i sont deux à deux distinctes.

On considère l'exécution de $\mathcal{B}$ sur w. Cette exécution existe et est unique car l'automate $\mathcal{B}$ est complet et déterministe. On note $q_1 \dots q_{n+1} q_\$ q'_1 \dots q'_{n+1}$ la suite des états correspondants et $\gamma_1 \dots \gamma_{n+1} \gamma_\$ \gamma'_1 \dots \gamma'_{n+1}$ le mot de Γ^* résultant de cette exécution. Puisque $n + 1$ est strictement supérieur au nombre d'états de $\mathcal{B}$, il existe deux entiers i et j avec $1 \leq i < j \leq n + 1$ tels que $q_i = q_j$. On considère le mot de données v obtenu à partir de w en échangeant les valeurs de données des de ces deux positions i et j. On obtient ainsi les deux mots w et v.

$$w = (a, d_1)\dots(\mathbf{a}, \mathbf{d_i})\dots(\mathbf{a}, \mathbf{d_j})\dots(a, d_{n+1})(\$, d_0)(a, d_1)\dots(a, d_{n+1})$$

$$v = (a, d_1)\dots(\mathbf{a}, \mathbf{d_j})\dots(\mathbf{a}, \mathbf{d_i})\dots(a, d_{n+1})(\$, d_0)(a, d_1)\dots(a, d_{n+1})$$

Par construction, on a bien $w \notin L_{\not\subseteq}$ et $v \in L_{\not\subseteq}$. Il reste maintenant à prouver que ces deux mots sont indistingables par l'automate $\mathcal{A}$.

Notons tout d'abord que $\gamma_i = \gamma_j$. En effet, $q_i = q_j$, les positions i et j du mot w sont étiquetées par la même lettre a et le transducteur $\mathcal{B}$ est déterministe. L'exécution de $\mathcal{B}$ sur w est aussi l'exécution de $\mathcal{B}$ sur le mot v. En effet, l'exécution de $\mathcal{B}$ sur un mot dépend uniquement de sa projection sur l'alphabet fini et les mots w et v ont la même projection sur l'alphabet fini. On obtient donc en particulier les mêmes mots de sortie si on considère la projection sur l'alphabet fini. Puisque $\gamma_i = \gamma_j$, la réécriture par $\mathcal{B}$ de chaque classe d_k (pour $0 \leq k \leq n + 1$ est la même dans w et dans v. On en déduit que l'automate de classe $\mathcal{C}$ accepte la réécriture de la classe d_k du mot w si et seulement si il accepte le réécriture de la même classe d_k dans le mot v et cela pour tous les k. Par définition de l'acceptance des automates à données, w est accepté par l'automate $\mathcal{A}$ si et seulement si v est accepté par $\mathcal{A}$.

Les deux mots w et v sont donc indistingables par l'automate $\mathcal{A}$ $\square$

Dans la suite, on utilise le terme *renommage* pour définir un morphisme lettre à lettre, c'est-à-dire un morphisme $h : \Sigma \to \Sigma'$ où Σ et Σ' sont des alphabets. Etant donné un mot de données w, l'image de ce mot par un morphisme h est le mot obtenu en appliquant le morphisme sur le projection sur l'alphabet fini de w. En particulier les mots $h(w$ et w ont exactement la même structure de données.

Lemme 2.35 Les langages reconnus par les automates à données sont clos par union, intersection et renommage.

Preuve. La clôture par union et intersection s'obtient de la manière classique. La clôture par renommage s'obtient simplement en utilisant un automate de base non déterministe.

On considère deux automates à données $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ et $\mathcal{A}' = (\mathcal{B}', \mathcal{C}')$. On note respectivement $L_\mathcal{A}$ et $L_{\mathcal{A}'}$ les langages correspondants, Γ et Γ' les alphabets de sorties des transducteurs $\mathcal{B}$ et $\mathcal{B}'$. Sans perte de généralité, on peut supposer les alphabets Γ et Γ' disjoints. De la même manière, on suppose que l'ensemble des états des différents automates sont disjoints.
On note $\mathcal{B} = (Q, Q_I, Q_F)$ où Q est l'ensemble des états ; Q_I (resp. Q_F) désigne le sous-ensemble des états initiaux (resp. finaux) et $\Delta \subseteq Q \times \Sigma \times \Gamma \times Q$ est

l'ensemble des transitions. De la même manière, on note $\mathcal{B}' = (Q', Q'_I, Q'_F, \Delta')$ avec $\Delta' \subseteq Q' \times \Sigma \times \Gamma' \times Q'$.
On note $\mathcal{C} = (S, S_I, S_F, \delta)$ où S est l'ensemble des états ; S_I (resp. S_F) désigne le sous-ensemble des états initiaux (resp. finaux) et $\delta \subseteq S \times \Gamma \times S$ est l'ensemble des transitions. Enfin, on note $\mathcal{C}' = (S', S'_I, S'_F, \delta')$ avec $\delta' \subseteq S' \times \Gamma' \times S'$.

Union. On définit l'automate $\mathcal{A}_\cup = (\mathcal{B}_\cup, \mathcal{C}_\cup)$ qui reconnaît l'union des langages $L_\mathcal{A}$ et $L_{\mathcal{A}'}$. L'automate de base est défini comme l'union de $\mathcal{B}$ et $\mathcal{B}'$: $\mathcal{B}_\cup = (Q \cup Q', Q_I \cup Q'_I, Q_F \cup Q'_F, \Delta \cup \Delta')$, l'alphabet de sortie du transducteur est donc $\Gamma \cup \Gamma'$. De la même façon, l'automate de classe est l'union des automates $\mathcal{C}$ et $\mathcal{C}'$: $\mathcal{C}_\cup = (S \cup S', S_I \cup S'_I, S_F \cup S'_F, \delta \cup \delta')$.

Par définition, un mot de données $w \in \Sigma \times \mathcal{D}$ est accepté par l'automate $\mathcal{A}_\cup$ si et seulement s'il existe une exécution acceptante de $\mathcal{B}_\cup$ sur $Str(w)$ qui écrit en sortie le mot $b_1 \ldots b_{|w|} \in (\Gamma \cup \Gamma')^*$, tel que pour chaque classe du mot w représentée par les indices de positions $i_1 < \cdots < i_k$, l'automate de classe $\mathcal{C} \cup \mathcal{C}'$ accepte le mot $b_{i_1} \ldots b_{i_k}$.

Puisque les alphabets Γ et Γ' ainsi que l'ensemble des états de transducteurs $\mathcal{B}$ et $\mathcal{B}'$ sont disjoints, le mot $b_1 \ldots b_{|w|}$ engendré par une execution acceptante de ce transducteur appartient soit à Γ^*, soit à Γ'^*. Ainsi, tous les mots $b_{i_1} \ldots b_{i_k}$ correspondant aux classes du mot w appartiennent à Γ^* dans le premier cas et à Γ'^* dans le second cas. Un mot de de données w est donc accepté par l'automate $\mathcal{A}_\cup$ si et seulement s'il existe :

- soit une exécution acceptante de $\mathcal{B}_\cup$ sur $Str(w)$ qui écrit en sortie le mot $b_1 \ldots b_{|w|} \in \Gamma^*$, tel que pour chaque classe du mot w représentée par les indices de positions $i_1 < \cdots < i_k$, l'automate de classe $\mathcal{C}_\cup$ accepte le mot $b_{i_1} \ldots b_{i_k} \in \Gamma^*$;
- soit une exécution acceptante de $\mathcal{B}_\cup$ sur $Str(w)$ qui écrit en sortie le mot $b_1 \ldots b_{|w|} \in \Gamma'^*$, tel que pour chaque classe du mot w représentée par les indices de positions $i_1 < \cdots < i_k$, l'automate de classe $\mathcal{C}_\cup$ accepte le mot $b_{i_1} \ldots b_{i_k} \in \Gamma'^*$.

Les alphabets Γ et Γ' ainsi que l'ensemble des états des automates $\mathcal{B}$, $\mathcal{B}$', $\mathcal{C}$ et $\mathcal{C}$' étant disjoints, le mot w appartient à $L_\mathcal{A}$ dans le premier cas et à $L_{\mathcal{A}'}$ dans le second cas. Les mots reconnus par l'automate $\mathcal{A}_\cup$ appartiennent donc à $L_\mathcal{A} \cup L_{\mathcal{A}'}$. Le fait que l'union des langages $L_\mathcal{A}$ et $L_{\mathcal{A}'}$ soit incluse dans le langage reconnu par $\mathcal{A}_\cup$ est immédiat.

Intersection. On définit maintenant l'automate $\mathcal{A}_\cap = (\mathcal{B}_\cap, \mathcal{C}_\cap)$ qui reconnaît l'intersection des langages $L_\mathcal{A}$ et $L_{\mathcal{A}'}$.
L'alphabet de sortie du transducteur $\mathcal{B}_\cap$ est $\Gamma \times \Gamma'$. L'ensemble des états de $\mathcal{B}_\cap$ est $Q \times Q'$, les états initiaux (resp. finaux) sont $Q_I \times Q'_I$ (resp. $Q_F \times Q'_F$). Les transitions sont définies comme l'ensemble des $((q, q'), a, (\gamma, \gamma'), (p, p'))$ tels que $(q, a, \gamma, p) \in \Delta$ et $(q', a, \gamma', p') \in \Delta'$. Le transducteur $\mathcal{B}_\cap$ simule en parallèle les deux transducteurs $\mathcal{B}$ et $\mathcal{B}'$ et engendre un mot bi-étiqueté par Γ et Γ'.
L'ensemble des états de l'automate de classe $\mathcal{C}_\cap$ est $S \times S'$, les états initiaux (resp. finaux) sont $S_I \times S'_I$ (resp. $S_F \times S'_F$). Les transitions de $\mathcal{C}_\cap$ sont définies comme l'ensemble des $((q, q'), (\gamma, \gamma'), (p, p'))$ tels que $(q, \gamma, p) \in \delta$ et $(q', \gamma', p') \in \delta'$. L'automate de classe simule sur la composante Γ (resp. Γ') le comportement de l'automate $\mathcal{C}$ (resp. $\mathcal{C}'$).

Par construction, l'automate $\mathcal{A}_\cap$ reconnaît exactement l'intersection des langages $L_\mathcal{A}$ et $L_{\mathcal{A}'}$.

Renommage. Étant donné un renommage $h : \Sigma \to \Sigma'$, on définit l'automate $\mathcal{A}_h = (\mathcal{B}_h, \mathcal{C})$ qui reconnaît le langage $h(L_{\mathcal{A}})$. L'alphabet de sortie du transducteur $\mathcal{B}_h$ est le même alphabet Γ que pour le transducteur $\mathcal{B}$.

Intuitivement, lorsque $\mathcal{B}_h$ lit une lettre de Σ', il devine quelle lettre de Σ a été renommée et procède comme pour le langage $L_{\mathcal{A}}$. Plus précisément, $\mathcal{B}_h = (Q, Q_I, Q_F, \Delta_h)$ où $\Delta_h \subseteq Q \times \Sigma' \times \Gamma \times Q'$ est l'ensemble des $(q, h(a), \gamma, p)$ tels que $(q, a, \gamma, p) \in \Delta$. A partir d'une exécution acceptante de $\mathcal{A}$ sur $w \in \Sigma$, il est facile de construire un chemin acceptant de $\mathcal{A}'$ sur $h(w)$, et vice versa. Les automates de classes étant identiques dans les deux automates cela suffit à montrer la correction de l'automate $\mathcal{A}_h$. □

Lemme 2.36 Les langages reconnus par les automates à données ne sont pas clos par complémentation.

Preuve. On appelle $L_{<}$ le langage complément du langage $L_{\not<}$ de la Remarque 2.32. Le langage $L_{<}$ ne peut être reconnu par aucun automate à données.

L'idée est la même que celle utilisée pour montrer que $L_{\not<}$ ne peut-être reconnu par un automate à données donc l'automate de base est déterministe.

Supposons que $L_{<}$ est reconnu par l'automate à données $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ dont l'automate de base $\mathcal{B}$ possède m transitions. On considère le mot de données $w = (a, d_1)(a, d_2) \dots (a, d_{m+1})\$(a, d_1)(a, d_2) \dots (a, d_{m+1})$ qui appartient au langage $L_{<}$. Il existe donc une exécution acceptante de $\mathcal{A}$ sur ce mot w. On note ρ cette exécution et $\rho_{\mathcal{B}}$ l'exécution correspondante de $\mathcal{B}$ sur w. Puisque $m+1$ est supérieur au nombre de transitions de $\mathcal{B}$, il existe deux positions $i, j \leq m+1$ correspondant à une même transition de $\mathcal{B}$ dans $\rho_{\mathcal{B}}$. Comme dans la preuve du Lemme 2.33, on construit v à partir de w en inversant les valeurs de données de ces deux positions i et j.

$$w = (a, d_1) \dots (\mathbf{a}, \mathbf{d_i}) \dots (\mathbf{a}, \mathbf{d_j}) \dots (a, d_{m+1})(\$, d_0)(a, d_1) \dots (a, d_{m+1})$$

$$v = (a, d_1) \dots (\mathbf{a}, \mathbf{d_j}) \dots (\mathbf{a}, \mathbf{d_i}) \dots (a, d_{m+1})(\$, d_0)(a, d_1) \dots (a, d_{m+1})$$

L'exécution ρ de $\mathcal{A}$ sur w est aussi une exécution acceptante de $\mathcal{A}$ sur v. Les mots w et v ont la même projection sur l'alphabet fini donc l'exécution $\rho_{\mathcal{B}}$ est aussi une exécution acceptante de $\mathcal{B}$ sur v. De plus par construction la réécriture de la classe d_k par $\rho_{\mathcal{B}}$ est la même dans w et dans v pour tout k.

Or, par construction, le mot v n'appartient pas au langage $L_{<}$. On obtient donc une contradiction. □

Lemme 2.37 Étant donné un langage régulier $L \subseteq \Sigma^*$, il existe un automate à données qui accepte les mots de données dont toutes les classes appartiennent au langage L.

Preuve. L'automate de base copie le mot et l'automate de classe teste l'appartenance au langage L. □

Remarque 2.38 Cet automate en lui-même ne semble pas très intéressant mais sa conjonction avec d'autres automates permet de vérifier des propriétés intéressantes. Par exemple à partir d'un tel automate, on peut vérifier l'appartenance à un langage régulier L pour certaines classes seulement en le combinant avec l'automate de l'exemple suivant.

Exemple 2.39 On veut construire un automate à données dont l'automate de base marque (de manière non déterministe) un nombre k de classes.

On procède de la manière suivante : l'automate de base a comme alphabet de sortie $\Gamma_k = \{\bot\} \cup (\{1, \dots, k\} \times \{0, 1\})$. Pour chaque position i, il choisit de manière non-déterministe un symbole $b_i \in \Gamma_k$ en assurant que pour chaque j, le symbole $(j, 1)$ est choisi exactement une fois. L'automate de classe accepte un mot s'il est dans $\bot^*$, ou $(j, 1)(j, 0)^*$, pour un certain j. Donc pour chaque classe, soit toutes les positions sont marquées du symbole $\bot$ par l'automate de base, soit la première position est marquée d'un $(j, 1)$ et toutes les autres $(j, 0)$. Comme un symbole $(j, 1)$ est utilisé exactement une fois, l'entier j identifie de manière unique une classe du mot de données.

L'automate de classe sait pour chaque position du mot si elle appartient à une classe marquée et si oui à laquelle. À partir de cet automate de classe, on peut construire un automate plus compliqué qui teste des propriétés sur les classes en fonction du marquage donné par l'automate de base.

Comparaison aux automates à registres

Les automates à données ont un plus grand pouvoir expressif que les automates à registres unidirectionnels présentés dans la partie précédente (voir la Définition 3.5).

Proposition 2.40 ([BS07]) Étant donné un automate à registres, on peut construire en temps polynomial un automate à données qui reconnaît le même langage.

Remarque 2.41 La réciproque est fausse. En effet, un automate à registre ne peut exprimer le fait que toutes les positions d'un mot sont dans des classes différentes. Cette propriété est très facile à vérifier à l'aide d'un automate à données en vérifiant à l'aide de l'automate de classe que chaque classe possède exactement un élément.

Ce résultat est intéressant car certaines propriétés s'expriment naturellement à l'aide d'un automate à registres. En voici un exemple :

Exemple 2.42 On considère un alphabet fini Σ augmenté par le prédicat unaire R ; c'est-à-dire l'alphabet $\Sigma' = \Sigma \times \{R, \overline{R}\}$. Étant donné un mot de données de $\Sigma' \times \mathcal{D}^*$, une position de ce mot est dite marquée par R si elle est étiquetée par une lettre de $\Sigma \times \{R\}$. Ce type de prédicat sera utilisé dans la suite pour distinguer des positions dans les mots de données.

Imaginons dans cet exemple que l'on veuille utiliser le prédicat S_{+1} afin de marquer exactement les positions x qui sont dans la même classe que leur successeur $x+1$. On peut vérifier cette propriété à l'aide de l'automate à registre $\mathcal{A}^r_+$ décrit ci-dessous. La Proposition 2.39 nous assure qu'il existe un automate à données qui vérifie la même propriété. Dans la suite, on appelle $\mathcal{A}_+$ cet automate à données.

L'automate $\mathcal{A}^r_+$ possède un registre r et quatre états q_0, q, q_+, $q_\bot$ tels que q_0 est l'état initial et les états q et q_+ sont acceptants. Dans l'état initial, l'automate mémorise la valeur de données courante et passe dans l'état q_+ si l'étiquette de la position courante appartient à $\Sigma \times \{S_{+1}\}$ et dans l'état q sinon. Au cours de l'exécution, lorsque l'automate est dans l'état q_+ (resp. q) il vérifie que la valeur de données courante est égale à (resp. différente de) la valeur du registre r puis mémorise la valeur de données courante et passe dans l'état $q_\perp$ si le test a échoué, q_+ si le test a réussi et si la position courante est marquée par S_{+1}, q sinon. L'état $q_\perp$ est un état puit dans lequel l'automate reste à partir du moment où il rencontre une position qui viole la propriété.

2.4.2 Décidabilité du vide

On montre maintenant que le vide des automates à données est décidable. Pour cela, on utilise les automates multicompteurs : étant donné un automate à données, on montre que l'on peut construire un automate multicompteurs qui reconnaît la projection sur l'alphabet fini du langage reconnu par l'automate à données (Proposition 2.50).

Les automates multicompteurs

Définition 2.43 Un **automate multicompteurs** est un automate fini non déterministe augmenté par un nombre fini n de compteurs. Un tel automate est de la forme $(Q, \Sigma, n, \delta, q_I, F)$ où Q est l'ensemble des états, Σ est l'alphabet d'entrée, n le nombre de compteurs, $q_I \in Q$ l'état initial, $F \subseteq Q$, l'ensemble des états finaux et δ la fonction de transition.

La relation de transition δ est un sous-ensemble de $Q\times(\Sigma\cup\{\epsilon\})\times\{Inc(i),\ Dec(i)\}_{i\leq n}$ $\times\, Q$. L'idée est qu'à chaque étape, l'automate peut changer d'état et modifier la valeur de ses compteurs en les incrémentant ou les décrémentant selon l'état courant et la lettre courante[2]. Si l'automate tente de décrémenter un compteur nul, l'éxecution bloque et le mot est rejeté.

Une **configuration** d'un automate multicompteurs est un tuple $(q, (c_j)_{j\leq n})$, où $q \in Q$ est l'état courant et $c_j \in \mathbb{N}$ représente la valeur du compteur j. Une **transition** $(p, a, Inc(i), q) \in \delta$ peut s'appliquer dans la configuration $(p, (c_j)_{j\leq n})$ si l'étiquette de la position courante est a. Après cette transition, l'automate est dans l'état q et la valeur des compteurs est inchangée sauf celle du i-ième compteur qui vaut $c_i + 1$. De la même manière, une transition $(p, a, Dec(i), q) \in \delta$ avec $a \in \Sigma\cup\{\epsilon\}$ peut s'appliquer dans la configuration $(p, (c_j)_{j\leq n})$ si l'étiquette de la position courante est a (lorsque $a \in \Sigma$) et si $c_i > 0$. Après cette transition, l'automate est dans l'état q et la valeur des compteurs est inchangée sauf celle du i-ième compteur qui vaut $c_i - 1$. Une **exécution** sur un mot $w \in \Sigma^*$ est une suite de configurations qui respecte la relation de transition δ. Une exécution est acceptante si elle commence dans l'état q_I avec tous les compteurs nuls et finit dans une configuration où l'état est final et la valeur de tous les compteurs est nulle.

Remarque 2.44 Les transitions de l'automate ne dépendent pas de la valeur des compteurs. En particulier, ce modèle d'automate n'autorise pas de test à zéro. Cependant, grâce aux ϵ-transitions, on peut vérifier que la valeur d'un compteur est au moins égale à k en décrémentant le compteur k fois puis en l'incrémentant k fois.

[2] Les ϵ-transitions sont autorisées.

Remarque 2.45 Demander que la valeur de tous les compteurs soit nulle dans la configuration finale ne change rien à l'expressivité du modèle. En effet, on peut rajouter une transition non-déterministe qui décrémente tous les compteurs afin d'assurer cette propriété.

Remarque 2.46 Le modèle des automates multicompteurs est aussi connu dans la litérature sous le nom de VASS ("Vector Addition System with States"). Ce modèle d'automate est équivalent à celui des réseaux de Petri. En particulier, le vide des automates multicompteurs et le problème d'accessibilité dans les réseaux de Petri sont équivalents [HP79].

Le théorème suivant est un résultat difficile qui a été montré au début des années 80. La décidabilité du vide des automates à données s'obtient à l'aide de ce théorème en montrant que l'on peut réduire le vide de ces automates au problème du vide pour les automates multicompteurs.

Théorème 2.47 ([Kos82, May84]) *Le vide des automates multicompteurs est décidable.*

Des automates à données aux automates multicompteurs.

L'idée de la réduction des automates à données vers les automates multicompteurs est de voir les mots acceptés par un automate à données comme des mots obtenus par une opération de type "shuffle".

Définition 2.48 Un mot $v \in \Sigma^*$ est un **shuffle** des n mots $u_1, \ldots, u_n$ si et seulement si on peut colorier l'ensemble de ses positions avec n couleurs de telle sorte que pour toute couleur i, le sous-mot correspondant dans w est le mot u_i.
Étant donnés n langages réguliers $L_1, \ldots, L_n$, on note $Sh(L_1, \ldots, L_n)$, l'ensemble des shuffles obtenus à partir de n mots $u_1 \in L_1$, …, $u_n \in L_n$.
Étant donné un langage régulier $L \subseteq \Sigma^*$, on note Shuffle(L) l'ensemble des shuffles d'un nombre quelconque de mots de L.

Le lien entre l'opération de "shuffle" et les automates multicompteurs est le suivant :

Proposition 2.49 ([Gis81]) Étant donné un langage régulier L, le langage Shuffle(L) est reconnu par un automate multicompteurs de taille bornée par la taille d'un automate fini non déterministe qui reconnaît L.

On donne maintenant le résultat et la réduction proprement dite.

Proposition 2.50 Étant donné un automate à données, on peut construire un automate multicompteurs qui reconnaît la projection sur l'alphabet fini du langage reconnu par l'automate à données.

Preuve.
Par définition des automates à données, un mot $v = a_1 \ldots a_n \in \Sigma^*$ appartient à la projection sur l'alphabet fini du langage reconnu par un automate à données $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ si et seulement s'il existe une exécution acceptante de $\mathcal{B}$ sur v qui écrit le mot $b_1 \ldots b_n \in \Gamma^*$ tel que $b_1 \ldots b_n$ est le shuffle de mots du langage

accepté par l'automate de classe $\mathcal{C}$ (noté $L_\mathcal{C}$). D'après la Proposition 2.48, le langage Shuffle($L_\mathcal{C}$) est reconnu par un automate multicompteurs $\mathcal{M}_\mathcal{C}$ de la même taille que $\mathcal{C}$.

À partir de cet automate multicompteurs $\mathcal{M}_\mathcal{C} = (Q, \Gamma, n, \delta, q_I, F)$ et du transducteur $\mathcal{B} = (S, S_I, S_F, \Delta)$, on peut construire un automate multicompteur $\mathcal{M}$ qui reconnaît l'ensemble des mots de Σ^* correspondant à la projection sur l'alphabet Σ des mots de données acceptés par $\mathcal{A}$.

Un état de l'automate $\mathcal{M}$ est un triplet $(s, \gamma, q) \in S \times \Gamma \times Q$. Intuitivement γ représente la lettre engendrée par le transducteur à la position courante. L'ensemble des états initiaux est $S_I \times \Gamma \times \{q_I\}$. L'ensemble des états finaux est $S_F \times \Gamma \times F$. Les compteurs du nouvel automate $\mathcal{M}$ sont identiques à ceux de $\mathcal{M}_\mathcal{C}$. Une configuration de l'automate $\mathcal{M}$ est donnée par un état (s, γ, q), l'étiquette courante $a \in \Sigma$ et la valeur des n compteurs. Supposons que (s, a, γ, s') est une transition de $\mathcal{B}$ et $(q, \gamma, IncDec, q')$ est une transition de l'automate $\mathcal{M}_\mathcal{C}$ où $IncDec$ représente des opérations sur les compteurs, alors pour toute lettre $\gamma' \in \Gamma$ la transition suivante est une transition de l'automate $\mathcal{M}$:

$$(s, \gamma, q), a, IncDec, (s', \gamma', q')$$

L'idée est de deviner à chaque transition la lettre engendrée par une exécution de $\mathcal{B}$ à la position successeur et de la stoquer dans l'état. Les transitions de $\mathcal{M}$ vérifient que cette exécution de $\mathcal{B}$ est correcte et simule l'exécution de $\mathcal{M}_\mathcal{C}$ en utilisant la lettre ainsi stoquée pour la position courante. □

On a maintenant tous les éléments pour montrer le résultat suivant :

Théorème 2.51 *Le vide des automates à données est décidable.*

Preuve. Ce résultat est une conséquence directe de la Proposition 2.50 et du Théorème 2.47 □

2.5 Décidabilité : le fragment à deux variables

Dans cette section, on utilise les automates à données pour montrer la satisfaisabilité du fragment à deux variables de la logique FO sur les mots de données. On étudie ensuite la compléxité de deux fragments de cette logique. Enfin, on s'intéresse aux extensions naturelles de la logique FO^2.

2.5.1 FO^2 et automates à données : décidabilité

On montre ici que l'on peut transformer une formule FO^2 en un automate à données de taille doublement exponentielle qui reconnaît le même langage (Proposition 2.53). On déduit de cette transformation un premier résultat de décidabilité (Théorème 2.52).

Théorème 2.52 *Le problème de satisfaisabilité des formules de la logique* $\mathrm{FO}^2(\sim, <, +1)$ *est décidable sur les mots de données finis.*

Preuve. Ceci découle de la Proposition 2.53 et du Théorème 2.51. □

Proposition 2.53 Étant donnée une formule de $\mathrm{FO}^2(\sim, <, +1)$, il existe un automate à données de taille doublement exponentielle qui reconnaît le même langage. La construction de cet automate est effective et se fait en temps doublement exponentiel dans la taille de la formule.

La preuve de la Proposition 2.53 se découpe en deux parties : on commence par donner une forme normale pour les formules FO^2 puis on montre que le langage correspondant à une formule en forme normale peut être reconnu par un automate à données.

Lemme 2.54 ([GO99]) Toute formule FO^2 est équivalente à une formule de la forme

$$\exists R_1 \ldots R_n \big(\forall x \forall y\ \chi \wedge \bigwedge_i \forall x \exists y\ \chi_i\big)\ ,$$

où les R_i sont des prédicats unaires et les formules χ et χ_i sont des formules sans quantificateurs. La taille de cette nouvelle formule est linéaire dans la taille de l'ancienne formule. On parle de *forme normale de Scott*.

L'idée est d'utiliser des prédicats unaires afin de remplacer dans φ les sous-formules contenant exactement une variable libre et ainsi faire diminuer la profondeur de quantification. On peut ainsi transformer récursivement la formule en remplaçant à chaque étape une sous-formule de la forme $\exists x\ \theta(x, y)$ ou $\forall x\ \theta(x, y)$ où θ est sans quantificateur par un nouveau prédicat unaire.[3] Ainsi chaque prédicat R_i correspond à une sous-formule de φ contenant exactement une variable libre. Dans un modèle de la formule φ, ce prédicat marque toute position qui vérifie la sous-formule correspondante. Les formules χ et χ_i contiennent l'information nécessaire pour assurer la cohérence de ces prédicats. Pour plus de détails, le lecteur peut se référer à l'article de Grädel et Otto [GO99].

Voici un exemple de transformation :

Exemple 2.55 On considère l'alphabet Σ et la formule φ suivante :

$$\varphi := \forall x\ [\ a(x) \rightarrow [\ \exists y\ [[\ x < y\ \wedge\ b(y)] \rightarrow [\forall x\ y < x \rightarrow \underline{[\forall y\ y < x \rightarrow\ x \nsim y]}\ ]]\ ]\ ]$$

On commence par rajouter un prédicat $R_1(x)$ afin de remplacer la partie souslignée. On obtient un nouvel alphabet $\Sigma_1 = \Sigma \times \{0, 1\}$ et une nouvelle formule :

$$\begin{aligned} \varphi_1 := &\quad \forall x\ [\ a(x) \rightarrow [\ \exists y\ [[\ x < y\ \wedge\ b(y)] \rightarrow \underline{[\forall x\ y < x \rightarrow R_1(x)\ ]}]\ ]\ ] \\ \wedge &\quad \forall x\ R_1(x) \leftrightarrow [\forall y\ y < x \rightarrow\ x \nsim y] \end{aligned}$$

On rajoute ensuite un prédicat $R_2(x)$ afin de remplacer la sous-formule souslignée. On obtient un nouvel alphabet $\Sigma_2 = \Sigma \times \{0, 1\}^2$ et une nouvelle formule :

$$\begin{aligned} \varphi_2 := &\quad \forall x\ [\ a(x) \rightarrow [\ \exists y\ [[\ x < y\ \wedge\ b(y)] \rightarrow R_2(y)]\ ]\ ] \\ \wedge &\quad \forall x\ R_1(x) \leftrightarrow [\forall y\ y < x \rightarrow\ x \nsim y] \\ \wedge &\quad \forall x\ R_2(x) \leftrightarrow [\forall y\ x < y \rightarrow R_1(y)\ ] \end{aligned}$$

[3] Il faut bien sûr rajouter une formule afin d'assurer la correspondance entre ce nouveau prédicat et la sous-formule.

On a un nouvel alphabet $\Sigma' = \Sigma \times \{0,1\}^2$. On réarrange la formule et on obtient la formule en forme normale :

$$\begin{aligned}&\forall x \exists y\ [\ a(x) \rightarrow [[\ x < y\ \wedge\ b(y)] \rightarrow R_2(y)]\ ]\ ]\\&\wedge \forall x \forall y\ [R_1(x) \leftrightarrow [\ y < x \rightarrow\ x \nsim y]]\ \wedge\ [R_2(x) \leftrightarrow [\ x < y \rightarrow R_1(y)\ ]]\end{aligned}$$

Maintenant que l'on a une forme normale, revenons à la preuve de la Proposition 2.52. Puisque les automates à données sont clos par intersection et renommage, on va montrer que l'on peut construire un automate à données pour chaque formule de la forme $\forall x \forall y\ \chi$ ou $\forall x \exists y\ \chi_i$. On prouve ces résultats dans les Lemmes 2.54 et 2.55.

Alphabets vs Prédicats : La transformation de la formule en forme normale de Scott engendre une augmentation de la taille de l'alphabet. Les prédicats unaires ainsi introduits peuvent être vu comme des marquages (voir l'exemple 2.41). D'une manière générale un alphabet fini peut-être représenté à l'aide d'un nombre logarithmique de prédicats unaires, une lettre correspondant à un ensemble de prédicats. L'étiquette d'une position peut donc être vue comme un vecteur de bits décrivant l'ensemble des prédicats marquant cette position.

Dans la suite, les transformations des formules introduisent de nouveaux prédicats unaires. Il est donc naturel d'utiliser cette notion de marquage. On appelle **type** toute conjonction cohérente de prédicats unaires et de négations de prédicats unaires. Ces prédicats sont soit des lettres de l'alphabet de départ ($a \in \Sigma$), soit des prédicats unaires introduits par la transformation de la formule en forme normale de Scott (R_i). Un type n'utilise pas forcément tous les prédicats.

Cependant cette représentation a des conséquences en terme de complexité, les automates de données utilisent un étiquetage des positions par un alphabet fini et non à l'aide de prédicats unaires. Il faut donc être conscient du fait que lorsqu'un automate veut tester si un prédicat marque la position courante, on peut avoir besoin d'un nombre exponentiel (dans le nombre de prédicats) de cas décrivant toutes les lettres dont le bit correspondant au prédicat testé vaut 1.

Lemme 2.56 Pour toute formule FO^2 de la forme $\phi = \forall x \forall y\ \chi$, où χ est sans quantificateur, on peut construire un automate à données de taille doublement exponentielle qui reconnaît le même langage.

Preuve. On commence par mettre χ en forme normale conjonctive. On peut ainsi réécrire la formule ϕ comme une conjonction (potentiellement exponentielle) de formules de la forme :

$$\psi = \forall x \forall y \quad \big[\alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x,y)\big]\ \rightarrow\ \gamma(x,y) \tag{2.1}$$

où α et β sont des types apparaissant dans ϕ, $\delta(x,y)$ est soit $x \sim y$ soit $x \nsim y$, et $\gamma(x,y)$ est une disjonction de formules atomiques utilisant seulement $<$ et $+1$. Il reste à montrer comment reconnaître les mots de données vérifiés par une formule de ce type à l'aide d'un automate à données. Pour chaque sous-formule, on peut construire un automate à données de taille constante. En considérant l'intersection de tous ces automates pour chacune des conjonctions, on obtient un automate de taille doublement exponentielle qui reconnaît bien les mêmes modèles que la formule de départ ϕ. La preuve est une étude des différents cas possibles pour les formules δ et γ.

On commence par le cas le plus simple : $\delta(x, y)$ est $x \sim y$. La formule ψ porte sur des positions d'une même classe. En ce sens, on peut dire que la formule ψ décrit une propriété de la projection sur l'alphabet fini (étendu par les prédicats unaires) des classes d'un modèle. Afin de décrire correctement cette propriété, on utilise le prédicat S_{+1} présenté dans l'Exemple 2.42. Supposons donc un alphabet étendu par le prédicat S_{+1} qui est tel que une position est marquée du prédicat S_{+1} si et seulement si elle est dans la même classe que son successeur. Si on considère deux positions x et y d'un mot de données qui sont dans la même classe $\mathcal{C}$, $y = x + 1$ si et seulement si $S_{+1}(x)$ et si les positions correspondant à x et y sont consécutives dans le mot obtenu en ne considérant que la classe $\mathcal{C}$.

Revenons à la formule :

$$\psi = \forall x \forall y \quad \left[\alpha(x) \ \wedge \ \beta(y) \ \wedge \ x \sim y\right] \ \rightarrow \ \gamma(x, y)$$

Si on supprime la sous-formule $x \sim y$ et qu'on remplace dans $\gamma(x, y)$ les formules atomiques $y = x + 1$ par[4] $y = x + 1 \wedge S_{+1}(x)$, on obtient une nouvelle formule ψ_{classe} n'utilisant pas le prédicat $\sim$. Les modèles de ψ sont exactement les mots de données marqués par le prédicat S_{+1} (avec la sémantique usuelle), toutes les classes vérifient ψ_{classe}.

En présence du marquage correspondant à S_{+1}, la formule ψ s'exprime donc comme une contrainte régulière sur chacune des classes du modèle. On a décrit dans la preuve du Lemme 2.37 un automate à données qui teste si la projection sur l'alphabet fini de chaque classe appartient à un langage régulier donné. On appelle $\mathcal{A}_{classe}$ l'automate correspondant à ψ_{classe}. Dans l'Exemple 2.42, on a montré qu'il existe un automate à données $\mathcal{A}_+$ vérifie que le prédicat S_{+1} marque exactement les positions qui sont dans la classe de leur successeur. Enfin, on appelle h le renommage qui à une lettre d'un alphabet $\Gamma \times \{S_{+1}, \overline{S_{+1}}\}$ associe la lettre correspondante de l'aphabet Γ.

Le langage associé à ψ est $h(\mathcal{A}_+ \cap \mathcal{A}_{classe})$. D'après les propriétés de clôture des automates à données, ce langage est reconnu par un automate à données.

Si la formule $\delta(x, y)$ est $x \not\sim y$, on doit analyser séparément les différentes disjonctions possibles correspondant à $\gamma(x, y)$. Le fait que x et y soient dans des classes différentes impose en particulier que $x \neq y$. Si on considère les différentes disjonctions possibles correspondant à $\gamma(x, y)$, on remarque qu'il suffit de considérer les cas suivants (et les cas symétriques où x et y sont échangés)[5].

$$\textit{true}, \quad \textit{false}, \quad x < y, \quad x = y+1, \quad x \neq y+1, \quad x \leq y+1, \quad y = x+1 \vee x = y+1.$$

Le cas $\gamma(x, y) := \textit{true}$ n'impose aucune contrainte, on peut donc oublier les formules correspondantes.

Si $\gamma(x, y) := \textit{false}$, cela signifie que le mot de données ne peut avoir une position α et une position β dans des classes différentes. Il y a donc deux solutions : soit un des deux types n'apparaît pas (ce qui est facile à tester à l'aide de l'automate de base) ; soit il y a une seule classe qui contient toutes les positions α et cette classe contient aussi toutes les positions β. En utilisant l'idée de l'Exemple 2.39, l'automate de base marque au plus une classe et l'automate de classe vérifie qu'aucune position α ou β n'apparaît dans une classe non marquée.

[4] On remplace aussi $x = y + 1$ par $x = y + 1 \wedge S_{+1}(y)$

[5] On peut montrer par induction que chacune des disjonctions est équivalente à une de ces formules en tenant compte du fait que $x \neq y$.

Le cas $\gamma(x,y) := x < y$ se traite en utilisant les notations données dans l'Exemple 2.22. La formule ψ est vérifiée si et seulement si (a) il n'y a pas de position β après la position Llst_α du mot et (b) les positions β situées entre $\mathrm{Llst}_\alpha + 1$ et Lst_α sont dans la classe de la position Lst_α. L'automate de base devine et marque les deux classes contenant respectivement la position $\mathrm{Lst}_\alpha(w)$ et $\mathrm{Llst}_\alpha(w)$ et teste si les propriétés (a) et (b) sont vérifiées. L'automate de classe assure que le marquage est correct.

Le cas où $\gamma(x,y)$ est $x \leq y+1$ se traite de la même manière.

Lorsque $\gamma(x,y) := (x = y+1)$, la formule ψ est vraie si et seulement si on est dans un des cas suivants :

- Les positions α et β sont toutes dans une même classe.
- Il n'y a pas de position α (resp. β) dans le mot.
- Le mot contient exactement deux positions α et deux positions β réparties en deux classes contenant chacune unie position de chaque type. De plus chaque position β précède la position α de l'autre classe.
- Le mot contient exactement une position α et au plus une position β qui n'appartient pas à sa classe. Si cette position existe elle précède la position α. Toutes les autres positions β du mot sont dans la classe de la position α.
- Le mot contient exactement une position β et au plus une position α qui n'appartient pas à sa classe. Si cette position existe, elle est précédée de la position β. Toutes les autres positions α du mot sont dans la même classe que la position β.

L'idée importante de ce cas est que si le mot contient des positions α et β alors, il n'existe pas plus de deux classes contenant α ou β. Un automate à données marque ces classes et vérifie les propriétés de voisinage avec l'automate de base et l'automate de classe permet de vérifier la cohérence du marquage et les propriétés de cardinalité des classes.

Le cas où $\gamma(x,y)$ est $y = x+1 \lor x = y+1$, suit la même idée que précédemment. Si le mot contient α et β alors il y a au plus trois classes contenant les α et β. Une étude de cas détaillée amène à des propriétés proches du cas $\gamma(x,y) := (x = y+1)$. On note ces classes 1, 2 et 3 dans la suite, les différents cas sont décrits à l'aide de contraintes de cardinalité sur les positions α et β de chaque classe et/ou décrivant l'arrangement des positions à l'aide d'un motif de taille constante.

- Toutes les positions α et β sont dans la même classe.
- Les positions α et β du mot sont regroupées dans le sous-mot : $\begin{smallmatrix}\alpha & \beta & \alpha & \beta & \alpha \\ 1 & 3 & 2 & 1 & 3\end{smallmatrix}$.
- La classe 3 contient un nombre quelconque de positions α, et les autres positions α ou β du mot sont regroupées dans le sous-mot $\begin{smallmatrix}\beta & \alpha & \beta & \alpha & \beta \\ 3 & 1 & 3 & 2 & 3\end{smallmatrix}$
- Les positions α et β sont réparties dans les classes 1 et 2 dans les deux sous-mots $\begin{smallmatrix}\beta & \alpha & \beta \\ 2 & 1 & 2\end{smallmatrix}$ et $\begin{smallmatrix}\beta & \alpha & \beta \\ 1 & 2 & 1\end{smallmatrix}$.
- La classe 1 contient un nombre quelconque de positions β et les autres positions α et β du mot sont réparties dans le sous-mot $\begin{smallmatrix}\beta & \alpha & \beta \\ 2 & 1 & 2\end{smallmatrix}$.
- Les cas présentés ci-dessus présentent les motifs maximaux qui peuvent apparaître. La formule est aussi vraie si certaines positions des motifs sont étiquetés par des types différents de α et β. Dans ce cas, on n'impose pas de contrainte sur les classes de ces positions.
- Pour compléter la liste des cas possibles, il reste à considérer les cas symétriques où les rôles de α et β sont échangés.

Comme dans le cas précédent où $\gamma(x, y)$ est la formule $(x = y + 1)$, on peut construire un automate à données qui marque les classes et vérifie que l'on est bien dans une des situations précédentes.

Le cas $\gamma(x, y) := (x \neq y + 1)$ se résoud directement en utilisant le prédicat S_{+1}.

$\square$

On s'intéresse maintenant à la seconde partie de la formule en forme normale de Scott.

Lemme 2.57 Pour toute formule FO^2 de la forme $\phi = \bigwedge_j \forall x \exists y\ \chi_j(x, y)$, où χ_j est sans quantificateur, on peut construire un automate à données qui reconnaît le même langage et est de taille doublement exponentielle dans la taille de la conjonction et des formules χ_j. La construction est effective et se fait en temps doublement exponentiel.

Preuve. On considère une formule de la forme $\forall x \exists y\ \chi(x, y)$. On commence par réécrire χ comme la disjonction d'un nombre exponentiel de formules φ_i de la forme

$$\alpha_i(x) \wedge \beta_i(y) \wedge \delta_i(x, y) \wedge \epsilon_i(x, y)\ ,$$

où α_i, β_i sont des types, δ_i représente une contrainte (in)égalité de données entre les positions x et y et ϵ_i une contrainte de positionnement entre x et y du type $x + 1 < y$, $x + 1 = y$, $x = y$, $x = y + 1$ ou $x > y + 1$.

La deuxième étape consiste à éliminer la disjonction en ajoutant pour chaque conjonction un nouveau prédicat S_i. Intuitivement, si S_i et α_i sont vrais dans une position x qui vérifie α_i, alors il existe une position y qui vérifie $\beta_i(y) \wedge \delta_i(x, y) \wedge \epsilon_i(x, y)$. Plus formellement, on réécrit chaque $\forall x \exists y\ \chi$ comme :

$$\exists S_1\ \exists S_2 \ldots \quad (\forall x\ \bigvee_i S_i(x))\ \wedge\ \bigwedge_i \forall x \exists y\ \big(\alpha_i(x) \wedge S_i(x) \rightarrow (\beta_i(y) \wedge \delta_i(x, y) \wedge \epsilon_i(x, y))\big)$$

La partie non triviale de cette preuve est de construire un automate à données qui correspond à une formule ψ de la forme :

$$\forall x \exists y \quad \alpha(x) \rightarrow\ \big[\beta(y)\ \wedge\ \delta(x, y)\ \wedge\ \epsilon(x, y)\big] \tag{2.2}$$

Comme dans la preuve du lemme précédent, on montre que la propriété décrite par une sous-formule peut être vérifiée par une conjonction d'automates à données de taille constante dans la taille de la formule. Pour chaque formule de ce type, on donne un automate à données de taille constante qui reconnaît le même langage. On doit ensuite considérer l'intersection (de taille exponentielle) de tous les automates pour toutes les sous-formules. On obtient ainsi la taille doublement exponentielle mentionnée dans l'enoncée du lemme.

Le cas où $\delta(x, y)$ est $x \sim y$ se résout de la même manière que dans la preuve du Lemme 2.56 en utilisant le prédicat S_{+1} puis en vérifiant une condition régulière sur chaque classe.

On considère maintenant le cas $\delta(x, y)\ =\ x \not\sim y$. Comme dans la preuve du Lemme 2.56, cela implique que $x \neq y$. Il faut alors étudier les différents cas selon la formule ϵ.

- $\epsilon(x, y) := x > y + 1$. Dans ce cas, ψ exprime le fait que toute position α a une position β dans une classe différente à sa droite (mais sans être son successeur). Par définition, toute position α située avant $\mathrm{Llst}_\beta - 2$ est sûre de vérifier la propriété. Il suffit donc de vérifier : (a) à partir de la position $\mathrm{Lst}_\beta - 1$, le mot ne contient pas de position α ; et (b) toutes les positions α situées entre $\mathrm{Llst}_\beta - 1$ et $\mathrm{Lst}_\beta - 2$ ne sont pas dans la classe de la position Lst_β. Pour vérifier cela, on devine les classes de Lst_β et Llst_β puis on vérifie les propriétés (a) et (b).
- Le cas où $\epsilon(x, y)$ est $x + 1 = y$ se traite en utilisant le prédicat S_{+1}. Étant donné ce prédicat, ψ se résume à une propriété régulière sur les classes.
- Les autres sont des cas symétriques des précédents.

□

2.5.2 Complexité : Automates Multicompteurs

La preuve du résultat de décidabilité du Théorème 2.52 ne donne pas de borne de complexité au problème de satisfaisabilité de la logique FO^2. Par la Proposition 2.53 et sachant que l'automate multicompteurs construit dans la preuve de la Proposition 2.50 est de taille linéaire dans la taille de l'automate à données, on a le résultat suivant :

Le problème de satisfaisabilité de la logique FO^2 se réduit en temps doublement exponentiel au problème du vide pour les automates multicompteurs. Ce problème est équivalent en terme de complexité au problème d'accessibilité dans les réseaux de Pétri dont on ne connaît pas de borne élémentaire ; on ne sait même pas s'il est primitif récursif.

Réciproquement, on peut réduire en temps polynomial le problème du vide des automates multicompteurs au problème de satisfaisabilité pour la logique $\mathrm{FO}^2(\sim, <, +1)$. La meilleure borne inférieure pour le problème du vide des automates multicompteurs est ExpSpace [Lip76].

Théorème 2.58 *Le vide des automates multicompteurs se réduit en temps polynomial au problème de satisfaisabilité de la logique* $\mathrm{FO}^2(\sim, <, +1)$.

Preuve. Étant donné un automate multicompteurs $\mathcal{M}$, on code les exécutions de l'automate dans des mots de données. On construit une formule FO^2 qui reconnaît exactement le codage des exécutions acceptantes de $\mathcal{M}$. Le vide de l'automate est donc équivalent à la satisfaisabilité de la formule.

On note Q l'ensemble des états, n le nombre de compteurs et δ la fonction de transition de $\mathcal{M}$. On considère l'alphabet $\Sigma = Q \cup \{D_1, I_1, \ldots, D_n, I_n\}$. On code une exécution de l'automate par un mot de données dont la projection sur l'alphabet fini correspond à la suite des transitions correspondantes. Une transition $(p, Inc(k), q) \in \delta$ est représentée par le mot pI_kq et une transition $(p, Dec(k), q) \in \delta$ par pD_kq. Les données permettent de contrôler la valeur des compteurs afin d'assurer d'une part qu'on ne décrémente pas de compteur nul et d'autre part que la valeur des compteurs est nulle à la fin de l'exécution. Pour cela une valeur de données fait correspondre la décrémentation d'un compteur à une incrémentation antérieure du même compteur. Plus précisément, une classe qui contient une position I_k ou D_k contient exactement deux positions : une première position étiquetée I_k et une position étiquetée D_k située à droite de la première. Il y a ainsi une bijection entre les positions I_k et D_k telle que toute position D_k (décrément du compteur k) est associée à exactement une position I_k (incrément du même compteur) qui la précède ; cela interdit le décrément d'un compteur vide au court du run correspondant. Le mot contient pour chaque k autant de positions I_k que de positions D_k ; la valeur des compteurs est donc nulle à la fin du run correspondant.

On construit facilement, en temps polynomial, une formule FO^2 qui reconnaît le codage des exécutions acceptantes. La formule vérifie (1) que la projection sur l'alphabet fini du mot est une suite de transitions qui respecte les transitions de l'automate, commence dans un état final et finit dans un état final, et (2) que toute classe qui contient une opération sur les compteurs est de la forme $I_k D_k$. La propriété (1) se vérifie facilement à l'aide d'une formule $\mathrm{FO}^2(+1)$ et la propriété (2) se vérifie de la même manière que dans l'Exemple 2.37.

Par construction une telle formule décrit exactement les codages des exécutions acceptantes de l'automate. □

Afin d'obtenir des résultats de complexité plus abordables, il faut donc considérer des fragments de la logique FO^2. On montre que le problème de satisfaisabilité de $\mathrm{FO}^2(\sim,<)$ est NExpTime-complet. Pour le fragment $\mathrm{FO}^2(\sim,+1)$, le problème est doublement exponentiel et NExpTime-dur.

2.5.3 Le fragment ordre

On considère tout d'abord le fragment obtenu si on n'autorise pas la relation successeur dans les formules. La complexité du problème de satisfaisabilité diminue alors considérablement : on montre que le problème est NExpTime-complet.

Théorème 2.59 *Le problème de satisfaisabilité de* $\mathrm{FO}^2(\sim,<)$ *est* NExpTime-*complet.*

Ce résultat doit être comparé avec le résultat présenté dans [EVW02], qui montre que le problème de satisfaisabilité est NExpTime-complet pour la logique $\mathrm{FO}^2(<)$ sur les mots. La relation de comparaison de valeurs de données $\sim$ n'est pas considérée dans ce travail, qui porte sur les mots étiquetés par un alphabet fini. La borne supérieure du Théorème 2.59 est donc plus forte puisqu'elle considère des modèles avec données et une logique autorisant les tests d'égalité de données. La preuve de la borne inférieure est différente de celle présentée dans [EVW02]. En effet, le pouvoir expressif apporté par les données est tel que la preuve peut se faire sans utiliser de prédicat unaire. La preuve présentée dans [EVW02] utilise quant à elle un nombre arbitraire de prédicats unaires. Sans valeurs de données, le problème de satisfaisabilité de $\mathrm{FO}^2(<)$ est NP-complet si on fixe le nombre de prédicats unaires [WI07]. Cette distinction n'est pas vraie en présence du prédicat successeur : le problème de satisfaisabilité de $\mathrm{FO}^2(+1)$ reste NExpTime-complet avec un nombre constant de prédicats [EVW02]. On redonne ce résultat dans la Proposition 2.65.

Proposition 2.60 On suppose que l'alphabet Σ contient une seule lettre.
Le problème de satisfaisabilité pour les formules de $\mathrm{FO}^2(\sim,<)$ est NExpTime-dur.

Preuve. On montre ce résultat en faisant une réduction du problème de dominos appelé **origin-constrained tiling problem** [Für83].

Une instance de "**origin-constrained tiling problem** " est la donnée de (T, n, H, V, t) où
- T est un ensemble de dominos,
- n est un entier,
- $H, V \subseteq T \times T$ correspond aux relations horizontales et verticales entre les dominos,
- $w_T \in T^n$ est une suite de dominos de départ.

Le problème est de savoir s'il existe un pavage du carré de côté 2^n qui respecte H et V et qui contient la suite des dominos w_T dans le coin en haut à gauche.

On montre que pour toute instance I du problème de dominos, il existe une formule φ_I de $\mathrm{FO}^2(\sim, <)$ qui reconnaît exactement les codages des solutions pour l'instance I. On en déduit que la formule est satisfaisable si et seulement si cette instance du problème a une solution.

L'alphabet Σ ne contenant qu'une seule lettre, on omet dans la suite les étiquettes des positions des mots de données. On considère une instance $I = (T, n, H, V, t)$ avec $T = \{t_1, \ldots, t_k\}$. On commence par décrire le codage d'un pavage par un mot de données. Soit π une fonction de pavage π de $2^n \times 2^n$ dans T, on appelle W_π un mot de données correspondant au codage de ce pavage.

Le mot W_π contient $2^{2n} + 1$ classes différentes. L'idée est de représenter chaque position (i, j) du carré par une valeur de donnée différente. La classe supplémentaire est nécessaire pour des raisons techniques ; elle est utilisée comme marqueur. Dans la suite on appelle $d_{(i,j)}$ la classe correspondant à la position (i, j) et $d_\#$ la classe supplémentaire. Le mot W_π est de la forme

$$w_1 \ldots w_n v_1 \ldots v_n u_1 \ldots u_k$$

où les w_ℓ, v_ℓ et u_ℓ sont des mots de données. Chacun de ces mots est formé d'une position de la classe $d_\#$ suivie de positions appartenants aux classes $d_{(i,j)}$. En particulier on peut remarquer que la classe $d_\#$ se distingue comme étant la classe associée à la premmière position du mot W_π et possède exactement $2n+k$ positions qui correspondant exactement aux premières positions des mots w_ℓ, v_ℓ et u_ℓ. La partie $w_1 \ldots w_n$ contient le codage en binaire des coordonnées horizontales de chaque position de la grille. La partie $v_1 \ldots v_n$ contient le codage en binaire des coordonnées verticales de chaque position de la grille. Et enfin, la partie $u_1 \ldots u_k$ sont utilisés pour représenter les dominos associés à chaque position par le pavage π. Plus précisément, étant donnée une position (i, j) de la grille et le domino $\pi(i, j)$, la classe $d_{(i,j)}$ correspondante dans le codage vérifie :
- Pour tout ℓ, le ℓ-ième bit du codage binaire de i est égal à 1 si et seulement si il existe dans le mot w_ℓ au moins une position de la classe $d_{(i,j)}$.
- Pour tout ℓ, le ℓ-ième bit du codage binaire de j est égal à 1 si et seulement si il existe dans le mot v_ℓ au moins une position de la classe $d_{(i,j)}$.
- Si $\pi(i, j) = t_m$, alors le mot u_m contient un position de la classe $d_{(i,j)}$ et cette classe n'apparait dans aucun autre mot u_ℓ de $u_1 \ldots u_k$.

On peut remarquer que cette description ne garantie pas l'unicité du mot W_π. En effet, rien ne garantie l'ordre d'apparition des positions des différentes classes dans les sous-mots v_ℓ,w_ℓ ou u_ℓ. De plus rien n'interdit la présence de plusieurs positions d'une même classe dans un sous-mot v_ℓ,w_ℓ ou u_ℓ. Ce qui importe pour la correspondance entre une classe et la position de la grille et le domino est la présence ou l'absence de cette classe dans chacun des sous-mots. Ceci est volontaire. Ce flexibilité permet en effet de décrire facilement l'ensemble des codage à l'aide d'une formule.

Exemple 2.61 On prend $T = \{t_1, t_2, t_3\}$ et un pavage qui contient le domino t_1 aux positions $(0,0)$ et $(1,1)$, le domino t_2 à la position $(0,1)$, et le domino t_3 à la position $(1,0)$. Un codage possible de ce pavage est présenté ci-dessous. Les valeurs de données $1,2,3,4$ correspondent respectivement aux classes $d_{(0,0)}$, $d_{(1,0)}$, $d_{(0,1)}$ et $d_{(1,1)}$ mentionnées ci-dessus. La valeur de données 0 correspond à la classe $d_{\#}$.

$$\overbrace{0\ 2\ 4}^{w_1}\quad \overbrace{0\ 3\ 4}^{v_1}\quad \overbrace{0\ 1\ 4}^{u_1}\quad \overbrace{0\ 3}^{u_2}\quad \overbrace{0\ 2}^{u_3}$$

Comme on vient de l'expliquer, un même pavage peut avoir plusieurs codages car on impose uniquement l'absence ou la présence des données à l'intérieur des blocs. En particulier, à partir de ce codage on peut construire d'autres codages en changeant l'ordre des données ou en dupliquant les positions à l'intérieur des blocs. Voici quelques exemples alternatif de codage du pavage ci dessus.

$$\overbrace{0\ 4\ 2\ 4}^{w_1}\quad \overbrace{0\ 3\ 3\ 3\ 4}^{v_1}\quad \overbrace{0\ 4\ 1}^{u_1}\quad \overbrace{0\ 3\ 3}^{u_2}\quad \overbrace{0\ 2}^{u_3}$$

$$\overbrace{0\ 4\ 2\ 2\ 4}^{w_1}\quad \overbrace{0\ 4\ 4\ 3\ 3\ 3\ 3}^{v_1}\quad \overbrace{0\ 1\ 4\ 1\ 4\ 1}^{u_1}\quad \overbrace{0\ 3\ 3\ 3}^{u_2}\quad \overbrace{0\ 2}^{u_3}$$

Et bien-sûr l'ensemble $\mathcal{D}$ étant infini et muni de la seule opération d'égalité, on peut aussi changer les données utilisées :

$$\overbrace{9\ 5\ 2}^{w_1}\quad \overbrace{9\ 3\ 2}^{v_1}\quad \overbrace{9\ 1\ 2}^{u_1}\quad \overbrace{9\ 3}^{u_2}\quad \overbrace{9\ 5}^{u_3}$$

On s'intéresse maintenant à la formule φ_I $\mathrm{FO}^2(\sim,<)$ qui reconnaît les codages des pavages correspondant à une solution de l'instance I. La formule est la conjonction de formules qui assurent les propriétés suivantes :

1. La classe associée à la première position du mot apparaît exactement $2n+k$ fois. On appelle $d_{\#}$ la valeur de donnée correspondante.
2. Toute valeur de données présente dans le mot et différente de $d_{\#}$ apparaît exactement une fois dans un bloc de $u_1 \dots u_k$. (Cela permet d'associer un domino à chaque valeur.)
   Deux valeurs de données différentes n'apparaissent jamais exactement dans les mêmes sous-mots de $w_1 \dots w_n$ et $v_1 \dots v_n$. (Cela assure que chaque valeur de données correspond à un position différente de la grille.)
3. Le pavage ainsi décrit couvre toute la grille et respecte les relations H et V, ainsi que la contrainte d'origine donnée par w_T.

On vérifie la propriété (1) à l'aide de la formule donnée dans l'Exemple 2.24. On explique maintenant comment vérifier les propriétés (2) et (3). On note $debut^{\sim}(x)$ la formule qui assure que la position x est dans la classe de la première position du mot. La formule $\beta_i(x)$ assure que la classe de x apparaît dans le i-ième bloc.

$$debut^{\sim}(x) ::= \exists y \ debut(y) \wedge x \sim y$$

$$\beta_i(x) ::= \exists y \ x \sim y \wedge [\exists x \ [x > y \wedge debut^{\sim}(x) \wedge \phi_{2n+k-i}(x) \wedge \neg\phi_{2n+k+1-i}(x)]]$$

La formule $\phi_j(x)$ correspond à la formule de l'Exemple 2.24 qui assure que la classe de la position x possède j éléments à droite de x.
La première partie de la propriété (2) est vérifiée par la formule suivante :

$$\forall x \forall y \quad x \not\sim y \rightarrow \big(debut^{\sim}(x) \vee debut^{\sim}(y) \vee \bigvee_{i=1,\ldots,2n} \neg[\beta_i(x) \iff \beta_i(y)]\big)$$

La seconde partie se vérifie de la même manière. On construit une formule pour la propriété (3) à l'aide des techniques usuelles de l'arithmétique sur les positions codées en binaire. Par exemple les positions associées aux positions x et y sont dans la même colonne de la grille si et seulement les classes correspondantes apparaissent exactement dans les mêmes sous-mots w_ℓ de la partie $w_1 \ldots w_n$. Cela correspond à la formule suivante :

$$\bigwedge_{i=1..n} \big[\beta_i(x) \leftrightarrow \beta_i(y)\big]$$

De la même façon, la formule suivante assure que les positions correspondant aux classes de x et y sont dans des colonnes successives de la grille :

$$\bigvee_{i=1..n} \big[\neg\beta_i(x) \wedge \beta_i(y) \quad \wedge \quad \bigwedge_{j<i} [\beta_j(x) \iff \beta_j(y)] \quad \wedge \quad \bigwedge_{j>i} [\beta_j(x) \wedge \neg\beta_j(y)]\,\big]$$

De la même manière, on construit des formules qui assurent que deux positions appartiennent à la même ligne ou à deux lignes successives. À l'aide de ces formules, on construit une formule qui vérifie que le pavage est complet et une formule qui vérifie que les contraintes issues des relations H et V sont bien vérifiées. $\square$

Proposition 2.62 ([Dav04]) Le problème de satisfaisabilité pour $\mathrm{FO}^2(\sim,<)$ est NExpTime.

Preuve. On réduit la satisfaisabilité d'une formule $\mathrm{FO}^2(\sim,<)$ au problème de satisfaisabilité pour $\mathrm{FO}^2(<)$. D'après [EVW02] la satisfaisabilité de $\mathrm{FO}^2(<)$ est NExpTime.

Comme dans la preuve de la Proposition 2.53, on commence par transformer (en temps linéaire) une formule FO^2 en une formule en forme normale de Scott. On obtient une formule de la forme suivante où les R_i sont des prédicats unaires, n est linéaire dans la taille de la formule de départ et les formules χ et χ_i sont sans quantificateur.

$$\exists R_1 \ldots \exists R_n \ \big[\forall x \forall y \ \chi(x,y) \wedge \bigwedge_i \forall x \exists y \ \chi_i(x,y)\big]$$

On montre maintenant que si une formule en forme normale de Scott est satisfaisable, elle l'est par un mot de données contenant au plus 2^{n+2} classes où n est le nombre de prédicats unaires présents dans la formule. Le résultat est alors une conséquence des travaux présentés dans [EVW02]. En effet, on se ramène à un problème sans donnée en codant chacune des 2^{n+2} classes à l'aide de $n+2$ nouveaux prédicats unaires.

Étant donné un modèle w de la formule, on construit un nouveau modèle w' en gardant uniquement les positions de certaines classes. On note α un type complet, c'est à dire un type qui décrit la valuation de tous les prédicats à une position. Le nouveau mot w' est obtenu à partir de w en gardant pour chaque type complet α les classes correspondant à $\mathrm{Fst}_\alpha, \mathrm{Ffst}_\alpha, \mathrm{Lst}_\alpha, \mathrm{Llst}_\alpha$ (si ces positions existent). On supprime ainsi toutes les autres classes du mot w. La formule possède n prédicats donc il existe au plus 2^n types complets. Le nouveau mot w' contient donc au plus $4 \cdot 2^n = 2^{n+2}$ classes.

Il faut maintenant montrer que le nouveau mot w' est toujours un modèle de la formule. La sous-formule $\forall x \forall y\ \chi(x,y)$ est toujours vraie dans w', puisque c'est un sous-mot de w. En ce qui concerne la partie $\forall\exists$, on montre que les classes gardées dans w' permettent de garantir la formule. Dans le mot w chaque position x possède une position témoin y telle que $\chi_i(x,y)$ est vraie. Si on considère une position x dans w' et le témoin y correspondant à la position x dans w pour la formule χ_i. Si y est dans la même classe que x, il est conservé dans w' et donc peut toujours servir de témoin à x dans w'. Si y (de type α) est dans une autre classe, une des positions $\mathrm{Lst}_\alpha(w)$, $\mathrm{Llst}_\alpha(w)$, $\mathrm{Fst}_\alpha(w)$ ou $\mathrm{Ffst}_\alpha(w)$ peut aussi servir de témoin à x dans w et donc dans w'. En effet, au moins une de ces positions est dans une classe différente de x et positionnée de la même manière par rapport à cette position. □

2.5.4 Le fragment successeur

On s'intéresse maintenant au fragment de FO^2 correspondant aux formules ne contenant pas la formule atomique $x < y$. Comme pour le fragment $\mathrm{FO}^2(\sim,<)$, la complexité du problème de satisfaisabilité diminue de manière significative. L'expressivité de cette logique est plus faible. Si on voit la projection sur l'alphabet fini des modèles (modulo recoloriage) comme l'intersection d'une condition régulière et d'un shuffle de langage régulier, La condition régulière imposée à chaque classe est beaucoup plus simple que dans le cas de la logique $\mathrm{FO}^2(\sim,<,+1)$. On n'a alors plus besoin de toute l'expressivité des automates à données pour représenter ces langages. On peut utiliser des modèles plus simples, ce qui donne des bornes de complexité probablement inférieures au cas général.

Théorème 2.63 *Le problème de satisfaisabilité pour* $\mathrm{FO}^2(\sim,+1)$ *est* NEXPTIME-*dur et dans 3*NEXPTIME.

On commence par montrer la borne inférieure puis on définit le modèle d'automates utilisés pour représenter les langages. Enfin, on explicite la représentation des langages et le résultat de complexité qui en découle.

La borne inférieure.

Proposition 2.64 Le problème de satisfaisabilité de la logique $\mathrm{FO}^2(\sim, +1)$ est NExpTime-dur.

Pour prouver la borne inférieure du Théorème 2.63, on montre un résultat plus fort car on montre que la borne inférieure peut être obtenue sans donnée. Ce résultat est mentionné dans [EVW02], on en redonne ici la preuve.

Proposition 2.65 ([EVW02]) Le problème de satisfaisabilité de la logique $\mathrm{FO}^2(+1)$ est NExpTime-dur.

Preuve. On utilise une réduction du problème de dominos présenté dans la preuve de la Proposition 2.60 *"origined-constraint tiling problem"*. Le codage utilisé est différent.

Étant donné une instance (T, n, H, V, w_T), on veut savoir s'il existe un pavage π du carré de côté 2^n par des dominos de l'ensemble T tel que le pavage respecte les relations [6] H et V et tel que la suite de n dominos donnée par w_T apparaît dans le coin en haut à gauche du carré.

Étant donnée une instance du problème $I = (T, n, H, V, w_T)$, on construit une formule de $\mathrm{FO}^2(+1)$ qui reconnaît exactement les codages des pavages solutions du problème.

On considère l'alphabet $\Sigma = T \cup \{\#, 0, 1\}$. On encode un pavage π par un mot du langage $(\#T\{0,1\}^{2n})^*\#$. L'idée est que chaque bloc $\#T\{0,1\}^{2n}$ représente une position du carré : la partie $\{0,1\}^{2n}$ correspond au codage en binaire des coordonnées dans le carré et la position T représente le domino correspondant du pavage π. On construit, en temps polynomial dans la taille de I, une formule $\mathrm{FO}^2(+1)$ de taille polynomiale dont les modèles sont les codages des solutions du problème de dominos pour l'instance I.

La formule vérifie les propriétés suivantes :

(1) Le mot appartient au langage $(\#T\{0,1\}^{2n})^*\#$.

(2) Chaque bloc correspond à une position différente du carré.

(3) Toutes les positions du carré apparaissent dans le mot.

(4) Le pavage respecte les relations H et V.

(5) La suite de dominos w_T apparaît en haut à gauche du carré.

En utilisant une formule basée sur celle de l'Exemple 2.25 ("Trouver un mot"), on vérifie facilement la propriété (1).

En utilisant la même idée, on construit les formules $Bit_i^{1\mathrm{H}}(x)$, $Bit_i^{0\mathrm{H}}(x)$, $Bit_i^{0\mathrm{V}}(x)$ et $Bit_i^{1\mathrm{V}}(x)$ pour $i \in \{1 \ldots n\}$. Ces formules expriment le fait le i-ième bit du codage de la coordonnée horizontale (resp. verticale) du domino correspondant est 1 ou 0. On utilise pour cela les formules $\phi_n^1(x)$ et $\phi_n^0(x)$ qui assurent respectivement que la position $x + n$ est étiquetée 1 ou 0.

$$\phi_0^1(x) ::= 1(x) \qquad \phi_n^1(x) ::= \exists y\ y = x + 1 \wedge \phi_{n-1}^1(y)$$

$$\phi_0^0(x) ::= 0(x) \qquad \phi_n^0(x) ::= \exists y\ y = x + 1 \wedge \phi_{n-1}^0(y)$$

[6] Ces relations décrivent respectivement les compatibilités horizontales et verticales entre les dominos.

$$Bit_i^{1\mathrm{H}}(x) ::= \#(x) \wedge \phi^1_{i+1}(x)$$
$$Bit_i^{1\mathrm{V}}(x) ::= \#(x) \wedge \phi^1_{i+n+1}(x)$$
$$Bit_i^{0\mathrm{H}}(x) ::= \#(x) \wedge \phi^0_{i+1}(x)$$
$$Bit_i^{0\mathrm{V}}(x) ::= \#(x) \wedge \phi^0_{i+n+1}(x)$$

La propriété (2) s'exprime maintenant de la manière suivante :

$$\forall x \forall y\, [x \neq y \wedge \#(x) \wedge \#(y)] \rightarrow \neg \bigwedge_{i=1..n} [Bit_i^{1\mathrm{H}}(x) \leftrightarrow Bit_i^{1\mathrm{H}}(y)] \wedge [Bit_i^{1\mathrm{V}}(x) \leftrightarrow Bit_i^{1\mathrm{V}}(y)]$$

Pour exprimer les propriétés (3-5), on définit les formules suivantes :

– les positions x et y sont dans la même colonne du pavage.

$$Colonne^{=}(x,y) := \#(x) \wedge \#(y) \bigwedge_{i=1..n} [Bit_i^{1\mathrm{V}}(x) \leftrightarrow Bit_i^{1\mathrm{V}}(y)]$$

– les positions x et y sont sur la même ligne du pavage.

$$Ligne^{=}(x,y) := \#(x) \wedge \#(y) \bigwedge_{i=1..n} [Bit_i^{1\mathrm{H}}(x) \leftrightarrow Bit_i^{1\mathrm{H}}(y)]$$

– la position y est dans la colonne à droite de la position x dans le pavage.

$$\begin{array}{rl} Colonne^{+1}(x,y) := \#(x) \wedge \#(y) \bigwedge_{i=1..n} & [\bigvee_{k<i} Bit_k^{0\mathrm{V}}(x)] \rightarrow Bit_i^{1\mathrm{V}}(x) \leftrightarrow Bit_i^{1\mathrm{V}}(y)] \\ \wedge \bigwedge_{i=1..n} & [\bigwedge_{k<i} Bit_k^{1\mathrm{V}}(x)] \rightarrow [Bit_i^{1\mathrm{V}}(x) \leftrightarrow \neg Bit_i^{1\mathrm{V}}(y)] \end{array}$$

– la position y est dans la ligne en dessous de la position x dans le pavage.

$$\begin{array}{rl} Ligne^{+1}(x,y) := \#(x) \wedge \#(y) \bigwedge_{i=1..n} & [\bigvee_{k<i} Bit_k^{0\mathrm{H}}(x)] \rightarrow Bit_i^{1\mathrm{H}}(x) \leftrightarrow Bit_i^{1\mathrm{H}}(y)] \\ \wedge \bigwedge_{i=1..n} & [\bigwedge_{k<i} Bit_k^{1\mathrm{H}}(x)] \rightarrow [Bit_i^{1\mathrm{H}}(x) \leftrightarrow \neg Bit_i^{1\mathrm{H}}(y)] \end{array}$$

On exprime la propriété (3) avec la formule suivante :

$$\begin{array}{cl} & \exists x\ \bigwedge_{i=1..n} \neg[Bit_i^{1\mathrm{H}}(x)\ \vee Bit_i^{1\mathrm{V}}(x)] \\ \wedge & \exists x\ \bigwedge_{i=1..n} [Bit_i^{1\mathrm{H}}(x)\ \wedge\ Bit_i^{1\mathrm{V}}(x)] \\ \wedge & \forall x\ \left[\#(x) \wedge \neg[\bigwedge_{i=1..n} Bit_i^{1\mathrm{H}}(x)]\right] \rightarrow [\exists y\ Colonne^{=}(x,y)\ \wedge\ Ligne^{+1}(x,y)] \\ \wedge & \forall x\ \left[\#(x) \wedge \neg[\bigwedge_{i=1..n-1} Bit_i^{1\mathrm{V}}(x)]\right] \rightarrow [\exists y\ Ligne^{=}(x,y)\ \wedge\ Colonne^{+1}(x,y)] \end{array}$$

La première (resp. deuxième) partie de la formule exprime le fait qu'il existe une position qui corespond au coin en haut à gauche (resp. en bas à droite) du carré. La fin de la formule exprime la continuité horizontale et verticale du codage entre ces deux positions.

On exprime maintenant le propriété (4), c'est à dire le fait que le pavage est correct :

$$\begin{array}{rl} \forall x \forall y & [Ligne^{=}(x,y)\ \wedge Colonne^{+1}(x,y)] \rightarrow H(x,y)) \\ \wedge & [Colonne^{=}(x,y)\ \wedge Ligne^{+1}(x,y)] \rightarrow V(y,x) \end{array}$$

où H et V représentent les contraintes entre les dominos. On exprime facilement ces formules comme une disjonction sur l'ensemble des paires de dominos.

Il reste à exprimer la propriété (5). On peut construire la formule correspondante de la même manière en utilisant les formules précédentes.

Bilan. Toutes ces formules sont encore de taille polynomiale en n. Par construction les modèles de la formule sont exactement les codages des solutions du problème de dominos pour l'instance I.

Le problème de satisfaisabilité pour $\mathrm{FO}^2(+1)$ est NExpTime-dur. $\square$

Remarque 2.66 Le résultat de la Proposition 2.64 reste vraie sans prédicat unaire[7] mais en utilisant les données. L'idée est de reprendre le codage utilisé dans la preuve du Lemme 2.65 et de le modifier légèrement afin de coder les prédicats unaires en binaire à l'aide des valeurs de données dans des blocs de taille constante. On dit qu'une position x est "étiquetée" 1 si elle est dans la même classe que son successeur et 0 sinon. À partir du codage de ces deux "étiquettes", on peut coder n'importe quel ensemble de prédicats unaires. Le codage de la preuve du Lemme 2.65 utilise un nombre fini m de prédicats unaires. On peut donc coder un prédicat dans un bloc de taille logarithmique m. On peut alors construire une formule qui identifie le bloc commençant à une position donnée comme étant le codage d'un prédicat. On peut aussi construire une formule qui vérifie que deux positions sont les débuts de deux blocs successifs du mot. On reprend alors les formules utilisées dans la preuve du Lemme 2.65 en remplaçant les formules atomiques $x = y + 1$ et les prédicats unaires par les nouvelles formules et on obtient une formule qui reconnaît les solutions encodées de cette nouvelle manière.

On s'intéresse maintenant à montrer la borne supérieure du Théorème 2.63.

La borne supérieure : idée de la preuve

L'absence de formule atomique du type $x < y$ diminue le pouvoir expressif des formules considérées. Le but de ce travail est de donner une représentation plus simple de la projection sur l'alphabet fini des langages reconnus par $\mathrm{FO}^2(\sim, +1)$ afin d'obtenir un meilleur résultat de complexité pour le problème de satisfaisabilité de ces formules. Dans le cas de la logique $\mathrm{FO}^2(\sim, +1, <)$, on montre que le langage est reconnu par un automate à données. On peut voir la projection sur l'alphabet fini de ces langages (modulo renommage) comme l'intersection d'un langage régulier (correspondant à l'automate de base) avec un langage de type shuffle de langage régulier (correspondant à l'automate de classe). Dans le cas de la logique $\mathrm{FO}^2(\sim, +1)$ les formules sont moins expressives. En particulier les contraintes imposées sur chaque classe sont beaucoup plus simples. On montre ici que l'on peut représenter la projection sur l'alphabet fini des modèles d'une formule de $\mathrm{FO}^2(\sim, +1)$ (modulo renommage) comme l'intersection d'un langage $\mathrm{FO}^2(+1)$ avec le shuffle de langages unitaires. Le résultat de complexité est une conséquence de cette représentation.

Définition 2.67 Un langage est dit **unitaire** s'il peut être représenté à l'aide de deux alphabets disjoints A et B comme l'ensemble des mots de $(A \cup B)^*$ qui contiennent exactement une occurrence de chacune des lettres de A.

On note $LC(A, B)$ ce langage.

[7] C'est à dire en considérant un alphabet réduit à une lettre.

On commence par montrer que l'on peut mettre les formules en forme normale. A partir de cette forme normale, on montre que la projection sur l'alphabet fini de ces langages peut s'exprimer à l'aide de langages unitaires. On montre ensuite les relations entre ces langages et des automates à contrainte linéaire. On a enfin tous les éléments pour conclure sur la complexité du problème de satisfaisabilité.

Mise en Forme Normale

On commence par donner deux formes normales plus précises que dans le cas général afin de traduire les formules de $\mathrm{FO}^2(\sim, +1)$.

Définition 2.68 Une formule est en forme normale intermédiaire si elle est de la forme $\exists R_1 \dots R_m\ (\theta_1 \wedge \cdots \wedge \theta_n)$, où les R_i sont des prédicats monadiques du second ordre et chaque formule θ_i est dans une des deux formes suivantes :

(1) $\forall x \forall y \quad \big[\rho(x,y)\ \wedge\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x,y)\big]\ \rightarrow\ \gamma(x,y)$

(2) $\forall x \exists y \quad \alpha(x) \rightarrow\ \big[\beta(y)\ \wedge\ \delta(x,y)\ \wedge\ \epsilon(x,y)\big]$

où

- α et β sont des types[8],
- $\rho(x,y) = (x \neq y+1 \wedge y \neq x+1 \wedge x \neq y)$,
- $\delta(x,y)$ est $x \sim y$ ou $x \not\sim y$,
- $\gamma(x,y)$ est $true$ ou $false$, et
- $\epsilon(x,y)$ est $x+1=y$, $x=y$, $x=y+1$ ou $\rho(x,y)$.

Définition 2.69 Une formule en *forme normale de données* est une formule de la forme :

$$\exists R_1 \dots R_m R_\# \bigwedge_\theta \theta,$$

où les R_i sont des prédicats unaires et chaque formule θ exprime une des propriétés suivantes (α, β représentent des types) :

(reg) θ est une formule de $\mathrm{FO}^2(+1)$.

($uniq$) Chaque classe contient au plus une occurrence de α.

($\alpha|\beta$) Les deux types α et β n'apparaissent pas dans une même classe.

($\alpha \subseteq \beta$) Toute classe qui contient une occurrence de α contient aussi une occurrence de β.

(#) $R_\#$ marque les positions dont le prédécesseur appartient à une classe différente.

On appelle ces propriétés des formules simples.

La première étape de la preuve consiste à montrer la proposition suivante.

Proposition 2.70 Étant donnée une formule ϕ de $\mathrm{FO}^2(\sim, +1)$, on peut construire une disjonction de formules en forme normale de données qui reconnaît le même langage.

$$\bigvee_i \exists R_1 \dots \exists R_\ell R_\# \psi_i$$

[8] On rappelle qu'un **type** est une conjonction cohérente de prédicats unaires et de négation de prédicats unaires.

Le nombre l de prédicats unaires et la taille de chacune des formules ψ_i sont exponentiels en la taille de ϕ et la disjonction est de taille doublement exponentielle dans la taille de ϕ.
Cette construction se fait en temps doublement exponentiel dans la taille de ϕ.

On montre successivement comment transformer une formule $\mathrm{FO}^2(\sim,+1)$ en une formule en forme normale intermédaire puis comment passer de cette forme normale à une formule en forme normale de données. Les résultats de la Proposition 2.70 découlent de ces deux transformations.

Lemme 2.71 Pour toute formule $\mathrm{FO}^2(\sim,+1)$, il existe une formule en forme normale intermédiaire qui reconnaît le même langage. Cette formule est au plus de taille exponentielle dans la taille de la formule de départ et contient un nombre lui aussi exponentiel de prédicats unaires R_i (variables du second ordre).

Preuve. Soit φ une formule de $\mathrm{FO}^2(\sim,+1)$. On commence par mettre la formule en forme normale de Scott (Lemme 2.54). On obtient donc une nouvelle formule équivalente de la forme :

$$\exists R_1 \dots R_n \big(\forall x \forall y\ \chi \wedge \bigwedge_i \forall x \exists y\ \chi_i\big)\ ,$$

où les R_i sont des prédicats unaires et les formules χ et χ_i sont des formules sans quantificateurs. Cette nouvelle formule est linéaire dans la taille de l'ancienne formule.

Transformation de la partie $\forall\forall$.

On considère maintenant la formule de la forme $\forall x \forall y\ \chi$. On peut réécrire cette formule comme :

$$\begin{array}{llcl} \forall x \forall y & \big(& y = x+1 & \to \psi_{y=x+1}(x,y) \\ & \wedge & x = y+1 & \to \psi_{x=y+1}(x,y) \\ & \wedge & x = y & \to \psi_{x=y}(x,y) \\ & \wedge & \rho(x,y) & \to \psi_{\rho}(x,y)\big) \end{array}$$

où les formules ψ sont sans quantificateurs et utilisent uniquement le prédicat binaire $\sim$ ainsi que les prédicats unaires. La taille de cette formule est de taille linéaire en la taille de χ. Comme les modèles de ces formules sont des mots (structure linéaire), on peut réécrire la formule de la manière suivante :

$$\begin{array}{ll} \forall x \forall y & \rho(x,y) \to \psi_\rho(x,y) \\ \wedge\ \forall x \exists y & \neg R_{fin}(x) \to (y = x+1\ \wedge \psi_{y=x+1}(x,y)) \\ \wedge\ \forall x \exists y & \neg R_{debut}(x) \to (x = y+1\ \wedge \psi_{x=y+1}(x,y)) \\ \wedge\ \forall x \exists y & x = y\ \to \psi_{x=y}(x,y) \end{array}$$

où R_{fin} et R_{debut} sont des prédicats unaires marquant respectivement la dernière et la première position d'un mot. Les trois sous-formules $\forall\exists$ seront traitées avec le cas $\forall\exists$. Il reste à traiter la sous-formule $\forall x \forall y\ \big(\rho(x,y) \to \psi_\rho(x,y)\big)$. On met ψ_ρ en forme normale conjonctive (modulo une explosion exponentielle de la taille de la formule) et on la réécrit comme :

$$\bigwedge_{\alpha,\beta} (\alpha(x) \wedge \beta(y)) \to \psi(x,y)$$

où α, β sont des types apparaissant dans ψ_ρ, et $\psi(x, y)$ est une formule sans quantificateur utilisant seulement $\sim$. Enfin, on réécrit $\psi(x, y)$ comme :

$$\psi(x,y) = (x \sim y \rightarrow \gamma_1) \wedge (x \nsim y \rightarrow \gamma_2)$$

où γ_1 et γ_2 sont égales à $false$ ou $true$.

On a ainsi transformé la formule $\forall x \forall y\ \chi$ en une formule équivalente de la forme suivante (la première conjonction étant de taille exponentielle dans la taille de la formule).

$$\begin{array}{ll}
\bigwedge_{\alpha,\beta} \forall x \forall y & \big[\rho(x,y)\ \wedge\ \alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x,y)\big]\ \rightarrow\ \gamma(x,y) \\
\wedge\ \forall x \exists y\, \chi_{+1} & \\
\wedge\ \forall x \exists y\, \chi_{-1} & \\
\wedge\ \forall x \exists y\, \chi_{=} &
\end{array}$$

Transformation de la partie $\forall\exists$.

On montre maintenant que chacune des formules de la forme $\forall x \exists y\ \chi$ peut être réécrite en une formule équivalente de la forme $\exists R'_1 \dots \exists R'_n \bigwedge_i \theta_i$ où θ_i est de type (2) (voir Définition 2.68). On montre de plus que la taille de chaque θ_i est polynomiale dans la taille de la formule de départ et le nombre n de prédicats est exponentiel dans taille de la formule.

On commence par réécrire χ comme la disjonction d'un nombre exponentiel de formules φ_j de la forme

$$\alpha_j(x) \wedge \beta_j(y) \wedge \delta_j(x,y) \wedge \epsilon_j(x,y)\ ,$$

où α_j, β_j sont des types, δ_j représente une contrainte (in)égalité de données entre les positions x et y et ϵ_j une contrainte de positionnement entre x et y (voir la définition de (2). Il ne reste plus qu'à éliminer la disjonction. Pour cela, on ajoute pour chaque formule de la disjonction ci-dessus un nouveau prédicat unaire $R_{\chi,j}$. Intuitivement le prédicat $R_{\chi,j}$ est vrai à la position x s'il existe un y tel que $\varphi_j(x, y)$ est vraie. Plus formellement, on réécrit chacune des formules $\forall x \exists y\ \chi$ comme :

$$\exists R_{\chi,1}\ R_{\chi,2} \dots \bigwedge_j \forall x \exists y\ \big(\alpha_j(x) \wedge R_{\chi,j}(x) \rightarrow (\beta_j(y) \wedge \delta_j(x,y) \wedge \epsilon_j(x,y))\big)\ .$$

En regroupant ainsi les réécritures de toutes les formules, on obtient bien une formule en forme normale intermédiaire. □

Lemme 2.72 Étant donnée une formule en forme normale intermédiaire Φ, on peut construire une disjonction de formules en forme normale de données qui reconnaît le même langage. La disjonction est de taille exponentielle dans la taille de Φ et chacune des formules en forme normale de données est de taille polynomiale dans celle de Φ.

Preuve. On considère une formule en forme normale intermédiaire :

$$\exists R_1 \dots R_m\ (\theta_1 \wedge \dots \wedge \theta_n)$$

On note T l'ensemble des types α, β qui apparaissent dans les formules θ_i. La taille de T est linéaire en n et m.

Information de classe.

La construction de la formule en forme normale de données s'obtient en réécrivant chaque θ_i en une combinaison booléenne de formules simples. Cette réécriture passe par une étude de cas qui dépend essentiellement du nombre de classes différentes qui contiennent un type α (à savoir si ce nombre est strictement inférieur à quatre ou supérieur à quatre). On ajoute donc des prédicats unaires afin de coder l'information nécessaire. La nouvelle formule commencera par une disjonction de toutes les situations possibles. On note que T étant de taille linéaire dans la taille de l'entrée, il y a un nombre exponentiel de situations à considérer.

Pour chaque type $\alpha \in T$ on définit quatre nouveaux prédicats unaires $R_1^\alpha, \ldots, R_4^\alpha$. Si le type α apparaît dans exactement k classes différentes avec $k \leq 4$ alors ces classes seront respectivement marquées par $R_1^\alpha, \ldots, R_k^\alpha$. Si le type α apparaît dans plus de quatre classes, on marque quatre classes à l'aide de $R_1^\alpha, \ldots, R_4^\alpha$. Pour des raisons techniques, on définit aussi quatre prédicats unaires $S_1^\alpha, \ldots, S_4^\alpha$. Chaque S_i^α, $1 \leq i \leq 4$, sert à marquer (s'il en existe une) une position de type $\alpha \wedge R_i^\alpha$ dans le mot.

Utiliser ces prédicats unaires va simplifier la suite du travail, il faut cependant que la formule assure que ces prédicats sont cohérents avec la définition qu'on en a donné. On appelle *fonction d'information de classe*, une fonction Occ : $T \to \{0, \ldots, 4, +\}$ qui associe à chaque type α de T le nombre de classes qui contiennent ce type $(0, 1, 2, 3, 4$ ou plus). Les prédicats $R_1^\alpha, \ldots, R_4^\alpha$, respectent Occ si :

(i) L'ensemble des positions marquées par R_i^α correspond exactement à une classe si $i \leq \mathrm{Occ}(\alpha)$ et est vide sinon.

(ii) L'ensemble des positions marquées par R_i^α contient une occurrence de α si $i \leq \mathrm{Occ}(\alpha)$.

(iii) Il existe des positions α non marquées par un R_i^α si et seulement si $\mathrm{Occ}(\alpha) = +$.

(iv) Pour chaque α, les ensembles de positions marquées par les R_i^α sont deux à deux disjoints.

Toutes ces conditions peuvent être exprimées à l'aide d'une conjonction de formules simples. Les conditions (ii) à (iv) s'expriment à l'aide de formules (*reg*). La condition (i) s'exprime de la manière suivante : aucune classe ne contient R_i^α et $\neg R_i^\alpha$ (formule simple de type $(\alpha|\beta)$) et toutes les occurrences de R_i^α sont dans la même classe. Cette propriété se vérifie en utilisant le prédicat S_i^α : une formule (*reg*) assure qu'il existe une occurrence de S_i^α dans le mot si et seulement si $i \leq \mathrm{Occ}(\alpha)$ et que cette position est unique et vérifie aussi α et R_i^α. Enfin, on demande que toute classe qui contient R_i^α contienne aussi S_i^α (formule simple de type $(\alpha \subseteq \beta)$). Cela assure que toutes les positions R_i^α sont dans la même classe.

Marquage des changements de classes.

On commence par introduire un nouveau prédicat unaire $R_\#$ qui va marquer les changements de classe dans le mot. Plus précisement, une position vérifie $R_\#$ si et seulement si elle n'est pas dans la même classe que la précédente position du mot ou c'est la première position du mot. Ce prédicat correspond exactement à la dernière formule simple $(\#)$.

On fixe une fonction d'information de classe. On montre maintenant comment transformer une formule θ en une conjonction de formules simples.

Transformation d'une formule de type (1) :

Supposons tout d'abord que θ est de la forme (1) :

$$\theta = \forall x \forall y \quad \big[(\rho(x,y) \ \wedge \ \alpha(x) \ \wedge \ \beta(y) \ \wedge \ \delta(x,y)) \ \rightarrow \ \gamma(x,y)\big]$$

- Si γ vaut $true$, la formule ne contraint en rien le modèle puisqu'elle est toujours vraie. On peut donc l'oublier.
- Si γ est $false$ et $\delta(x,y)$ est $x \sim y$, la formule impose que, si α et β apparaissent dans une même classe, alors ils sont dans des positions adjacentes ou égales. Pour une telle classe, il y a donc au plus trois positions α et β qui sont consécutives et alternées ($\alpha\beta\alpha$ ou $\beta\alpha\beta$). Si un des deux types apparaît dans moins de cinq classes, celles-ci sont marquées par les prédicats R_i, on peut donc exprimer la propriété correspondant à θ à l'aide d'une formule simple de type (reg). Si les deux types apparaissent dans plus de quatre classes, on introduit quatre nouveaux prédicats $R^\theta, R^\theta_{11}, R^\theta_{21}, R^\theta_{22}, R^\theta_{31}, R^\theta_{32}, R^\theta_{33}$ tels que :
  - une classe qui ne contient pas les deux types α et β est marquée par R^θ ;
  - si α et β apparaissent dans une classe à une même position, celle-ci est marquée R^θ_{11} ;
  - si α et β apparaissent dans une classe dans deux positions, les positions correspondantes sont marquées de gauche à droite par $R^\theta_{21}, R^\theta_{22}$;
  - si α et β apparaissent dans une classe dans trois positions, les positions correspondantes sont marquées de gauche à droite par $R^\theta_{31}, R^\theta_{32}, R^\theta_{33}$;
  - une classe contient au plus une occurrence de R^θ_i ;
  - une classe qui contient R^θ_{11} ne contient aucun autre R^θ_i ;
  - une classe qui contient R^θ_{21} contient aussi R^θ_{22} mais aucun autre R^θ_i et réciproquement ;
  - une classe qui contient R^θ_{3i} contient les deux autres R^θ_{3i} mais aucun autre R^θ_i ;
  - aucune position n'est marquée par R^θ et R^θ_i.

  Toutes ces propriétés peuvent être décrites par des formules simples de la même manière que les prédicats R^α_i introduits précédemment. On utilise le prédicat $R_\#$ pour assurer que les positions R^θ_i d'une même classe sont consécutives. On impose que toute position R^θ_{21} soit suivie d'une position R^θ_{22} qui ne vérifie pas $R_\#$; de la même façon on impose que toute position R^θ_{31} soit suivie d'une position R^θ_{32} puis d'une position R^θ_{33} et que ces deux positions ne vérifient pas $R_\#$.
- Si γ vaut $false$ et $\delta(x,y)$ est $x \nsim y$, la formule impose que, si les deux types apparaissent dans des classes différentes, alors ils doivent être dans des positions adjacentes ou égales. Si un des deux types n'apparaît pas dans le mot, cette formule est toujours vraie. On la remplace par $true$. Si le mot contient les deux types et qu'au moins un des deux apparaît dans quatre classes différentes, la formule ne peut être satisfaite ; on la remplace par $false$. Si les deux types apparaissent dans moins de quatre classes, celles-ci sont marquées. Comme dans le cas précédent, on peut exprimer les différentes configurations possibles à l'aide de formules (reg).

Transformation d'une formule de type (2) :

On suppose maintenant que θ est de type (2) :

$$\forall x \exists y \quad \alpha(x) \rightarrow \ \big[\beta(y) \ \wedge \ \delta(x,y) \ \wedge \ \epsilon(x,y)\big]$$

Si $\epsilon(x,y)$ est $x+1=y$, $x=y$ ou $x=y+1$ alors, on peut facilement exprimer θ à l'aide de formules (reg) en utilisant le prédicat $R_{\#}$. Il reste donc le cas où $\epsilon(x,y)$ est $\rho(x,y)$.

- Si $\delta(x,y)$ est $x \not\sim y$, la formule impose que pour toute position α il existe une position non adjacente[9] β dans une classe différente.
  - Si β apparaît dans au moins quatre classes différentes alors la formule est toujours vraie. On peut donc la remplacer par $true$.
  - Si β apparaît dans au plus trois classes, on peut exprimer la formule θ à l'aide de formules (reg) en utilisant les prédicats R_{β_i}.
- Si $\delta(x,y)$ est $x \sim y$, la formule impose que pour toute position α il existe une position dans la même classe qui n'est ni adjacente ni identique et vérifie β. Pour exprimer cette formule, on introduit des nouveaux prédicats unaires $R^\theta, R^\theta_{11}, R^\theta_{21}, R^\theta_{22}, R^\theta_{31}, R^\theta_{32}, R^\theta_{33}$ afin de distinguer le nombre d'occurrences de β dans chaque classe.
  - Une classe est étiquetée R^θ si et seulement si elle contient au moins quatre occurrences de β.
  - Si une classe contient une seule position β, elle est étiquetée R^θ_{11}.
  - Si une classe contient exactement deux positions β, elles sont étiquetées R^θ_{21} et R^θ_{22} (de gauche à droite).
  - Si une classe contient exactement trois positions β, elles sont étiquetées R^θ_{31}, R^θ_{32} et R^θ_{33} (de gauche à droite).

  Comme dans le cas $\gamma = false$, $\delta(x,y) = x \sim y$ du cas (1), on peut assurer que les prédicats vérifient les propriétés précédentes à l'aide de formules simples. Pour décrire complètement la formule θ, on utilise des formules (reg) et le prédicats $R_{\#}$ pour assurer qu'aucune position α n'est voisine ou identique à une positions β de sa classe.

Bilan : A partir d'une formule Φ en forme normale intermédiaire, on a construit pour chaque fonction d'information de classe une formule en forme normale de donnée de taille polynomiale dans la taille de la formule Φ. Le nombre de fonctions d'information de classe est exponentiel dans la taille de Φ. $\square$

De la Forme Normale aux Langages Unitaires

On considère une formule Φ en forme normale de données. On veut exprimer le langage correspondant à l'aide de langages unitaires. Une formule en forme normale de donnée est de la forme :

$\exists R_1 \ldots R_m R_{\#} \bigwedge_\theta \theta$ où les θ sont des formules simple de type (reg), $(uniq)$, $(\alpha|\beta)$, $(\alpha \subseteq \beta)$, $(\#)$.

On note T, l'ensemble des types apparaissant dans les formules simples et T_{uniq} l'ensemble des types apparaissant dans les formules simples de type $(uniq)$. Ces ensembles sont de taille linéaire dans la taille de la formule.

Modulo l'ajout d'un nombre lineaire de prédicats unaire et de formules simples, on peut supposer que tous les types apparaissant dans les formules $(\alpha \subseteq \beta)$

[9]Cette position doit aussi être non-identique mais cela est garanti par le fait qu'elle est dans une classe différente.

apparaissent aussi dans les formules ($uniq$). En effet si une formule de type ($\alpha \subseteq \beta$) impose que si une classe contient une position de type α alors elle contient aussi une position de type β et que $\alpha \notin T_{uniq}$, il suffit d'ajouter un prédicat unaire qui va servir à marquer au plus une position α par classe, on note α_0 le type correspondant. On remplace la formule simple ($\alpha \subseteq \beta$) par une formule qui impose que toute classe qui contient un α_0 contient un β et on ajoute la formule de type ($uniq$) pour le type α_0. Il est facile de voir que cela ne change rien aux modèles de la formule.

On dit qu'un sous ensemble A de T_{uniq} satisfait les formules de type ($\alpha|\beta$) et ($\alpha \subseteq \beta$) si (1) pour types α, β de A, aucune formule de type ($\alpha|\beta$) n'interdit la présence de α et β dans la même classe et (2) pour tout type α de A, si une formule ($\alpha \subseteq \beta$) impose la présence de β alors β est aussi un type de A. On appelle dans la suite A_i un tel ensemble.

On associe à chaque A_i un ensemble B_i qui représente l'ensemble des types de T qui satisfont[10] les formules ($\alpha|\beta$). Il existe au plus un nombre exponentiel de tels couples d'ensembles qui sont chacun de taille linéaire dans la taille de T.

On note L le langage régulier[11] correspondant aux formules de type (reg).

Un mot de données sur un alphabet étendu à tous les types T est un modèle de Φ si et seulement si :

(i) il respecte la contrainte régulière L ;

(ii) la projection sur l'alphabet fini de chaque classe appartient à un des langages LC(A_i, B_i) ;

(iii) le marquage (#) correspond bien aux changements de classes.

Les conditions (i) et (ii) sont équivalentes à dire que la projection sur l'alphabet fini du mot est dans le langage $L \cap \text{Shuffle}(\bigcup_i LC(A_i, B_i))$. La condition (iii) impose de raffiner ce langage. Dans la description de la projection sur l'alphabet fini, le marquage des classes se traduit comme le marquage des points de découpage du mot dans le shuffle.

Définition 2.73 On définit le **shuffle marqué** du langage L comme l'ensemble des mots w de Shuffle(L) pour lesquels il existe $w_1, \ldots, w_n \in L$ tels que $w \in Sh(w_1, \ldots, w_n)$ et une décomposition π telle que les positions étiquetées # dans le mot w marque exactement les positions x telles que $\pi(x) \neq \pi(x-1)$.
On parle de décomposition marquée par # et on note $\text{Shuffle}_{\#}(L)$, le shuffle marqué du langage L.

On verra dans la suite que cette propriété s'exprime à l'aide de morphisme et de la notion de shuffle de voisinage. Le lemme suivant découle des remarques précédentes.

Lemme 2.74 La projection sur l'alphabet fini d'un langage défini par une formule Φ en forme normale de données est de la forme $L \cap \text{Shuffle}_{\#}(\bigcup_i LC(A_i, B_i))$ où L est un langage FO^2(+1), le nombre de langages $LC(A_i, B_i)$ est exponentiel dans la taille de la formule Φ et les ensembles A_i et B_i sont de taille polynomiale dans la taille de Φ.

[10] Si on considère une paire de types $\{\alpha, \beta\}$ de l'ensemble $A_i \cap B_i$ aucune formule de type ($\alpha|\beta$) n'interdit la présence similtanée de ces deux types dans une même classe.

[11] Le langage L est un langage EMSO(+1). Cette classe de langages est égale à la classe des langages réguliers.

Langages unitaires et Automates à contraintes linéaires

On définit maintenant une nouvelle classe d'automates, notés LCA, qui vont nous permettre de représenter et manipuler les langages unitaires. On donne les définitions nécessaires puis on montre des propriétés de clôture et des résultats de complexité.

Une **contrainte linéaire sur les variables** Z est une combinaison booléenne d'inégalités linéaires :

$$\sum_{z\in Z} k_z \cdot z \quad \leq \quad \sum_{z \in Z} l_z \cdot z \qquad k_z, l_z \in \mathbb{N} \ .$$

Une valuation $\mu : Z \to \mathbb{N}$ satisfaisant la contrainte se définit de la manière usuelle. La *solution* d'une contrainte linéaire est l'ensemble des valuations qui la vérifie. On dit qu'un mot $w \in Z^*$ satisfait une contrainte linéaire si la valuation suivante satisfait la contrainte :

$$\mu_w : Z \to \mathbb{N} \ \text{ où } \mu_w(z) \text{ représente le nombre de positions } z \text{ dans } w$$

Définition 2.75 Un **automate à contrainte linéaire** (LCA) est un automate fini non déterministe associé à une contrainte linéaire sur l'ensemble Q de ses états. Un LCA $\mathcal{A}$ accepte un mot w si et seulement s'il existe une exécution acceptante $(q_0, a_1, q_1, \ldots, a_n, q_n)$ de $\mathcal{A}$ sur w, telle que $q_0 \ldots q_n$ satisfasse la contrainte linéaire. On dit qu'un langage est un langage LCA s'il peut être reconnu par un LCA.

On donne maintenant une série de propriétés sur ces automates.

Théorème 2.76 ([SSMH04]) *Le vide des LCA est* NP*-complet.*

Lemme 2.77 La classe des LCA est close par conjonction, disjonction et renommage.

Preuve. Il suffit de considérer les constructions standard correspondant aux automates finis, les contraintes se composent elles aussi facilement. □

Lemme 2.78 Étant donnés deux langages LCA L_1 et L_2, le langage $Sh(L_1, L_2)$ est aussi un langage LCA.

Preuve. On considère deux automates LCA $\mathcal{A}_1$ et $\mathcal{A}_2$ qui reconnaissent respectivement les langages L_1 et L_2. L'ensemble des états de l'automate LCA qui reconnaît $Sh(L_1, L_2)$ est le produit des états $\mathcal{A}_1$ et des états de $\mathcal{A}_2$. Au cours d'une exécution, l'automate devine une décomposition et choisit si la position courante est dans la partie correspondant au langage L_1 ou L_2. Il effectue alors la transition en conséquence. La contrainte linéaire se construit naturellement à partir des contraintes des deux automates. □

On appelle **substitution régulière** de Σ dans Γ un homomorphisme qui associe à toute lettre de l'alphabet Σ un langage régulier de Γ.

Lemme 2.79 Les langages LCA sont clos par substitution régulière.

Preuve. On considère un langage LCA $L \subseteq \Sigma^*$ et une substitution régulière h de Σ dans Γ^*. L'automate qui reconnaît $h(L)$ fonctionne ainsi : étant donné un mot $w \in \Gamma^*$, l'automate devine une décomposition $w_1 \in h(a_1), \dots, w_n \in h(a_n)$ telle que $w = w_1 \dots w_n$ et $a_1, \dots, a_n \in \Sigma$. L'automate simule alors l'automate correspondant à L sur la séquence devinée $a_1, \dots, a_n$.

Plus précisément, on note $\mathcal{A}$ un automate LCA qui reconnaît L, et Q l'ensemble de ses états. On considère pour chaque lettre $a \in \Sigma$ un automate $\mathcal{B}_a$ qui reconnaît le langage régulier $h(a)$. On note Q_h l'union (disjointe) des états des automates $\mathcal{B}_a$ pour $a \in \Sigma$.

On définit un automate LCA $\mathcal{A}_h$ qui reconnaît $h(L)$. L'ensemble des états de cet automate est $Q \times \Sigma \times Q_h \times \{0, 1\}$. Les états initiaux de l'automate sont $(q_i, a, s_i, 1)$ où q_i (resp. s_i) est un état initial de $\mathcal{A}$ (resp. B_a), a est une lettre de Σ. Au cours d'une exécution, si la lettre courante est $\gamma \in \Gamma$ et l'état (q, a, s, ϵ) où $\epsilon \in \{0, 1\}$ et s un état de $\mathcal{B}_a$, l'automate A_h peut passer dans un état $(q, a, s', 0)$ si (s, γ, s') est une transition de l'automate $\mathcal{B}_a$. Si s est un état final de $\mathcal{B}_a$ l'automate peut aussi passer dans un état $(q', a', s', 1)$ où (q, a, q') est une transition de $\mathcal{A}$, a' est une lettre de Σ et s' un état de $\mathcal{B}_{a'}$ tel qu'il existe une transition (s_i, γ, s') dans $\mathcal{B}_{a'}$ (où s_i est initial).

La contrainte linéaire de l'automate $\mathcal{A}_h$ se décrit en remplaçant $\#q$ pour $q \in Q$ par la somme des $\#(q, a, s, 1)$ pour $a \in \Sigma$ et $s \in B_h$.

La taille de $\mathcal{A}_h$ est polynomiale dans la taille de $\mathcal{A}$ et $\mathcal{B}_a$ (pour $a \in \Sigma$). $\square$

On s'intéresse maintenant aux shuffles de languages unitaires. On montre que les LCA peuvent reconnaître un shuffle spécial d'une union de langages unitaires (appelé *shuffles de voisinage*). On utilise ensuite les substitutions régulières pour étendre le résultat afin de reconnaître la projection sur l'alphabet fini des modèles d'une formule.

Définition 2.80 On définit le **shuffle de voisinage** de L comme l'ensemble des mots w de Shuffle(L) pour lesquels il existe $w_1, \dots, w_n \in L$ tels que $w \in Sh(w_1, \dots, w_n)$ et une décomposition π telle que $\pi(x) \neq \pi(x+1)$ pour toute position x de w exeptée la dernière.
On note NShuffle(L), le shuffle de voisinage du langage L.

Lemme 2.81 Si $L_1, \dots, L_k$ sont des langages unitaires, alors le langage NShuffle($L_1 \cup \dots \cup L_k$) est un langage LCA.

Preuve. On considère que chaque langage L_i correspond au langage LC(A_i, B_i) où A_i et B_i sont des alphabets deux à deux disjoints[12]. On présente maintenant un marquage (ou recoloriage) des mots de NShuffle($L_1 \cup \dots \cup L_k$) qui facilite la description du LCA reconnaissant le langage.

Marquage.
On choisit une constante m supérieure au cardinal de chacun des ensembles $A_i \cup B_i$. Pour des raisons techniques on impose que m est supérieur à trois.
On considère un mot de NShuffle($L_1 \cup \dots \cup L_k$) et une décomposition de celui-ci comme un shuffle du langage $L_1 \cup \ldots \cup L_k$. Pour chaque $i = 1, \dots, k$, on distingue deux situations : soit la décomposition utilise un nombre j de mots de L_i inférieur à m, alors on marque chacun de ces mots avec un marqueur spécial

[12]Cette hypothèse peut être omise et remplacée par une étape de renommage. On l'impose ici afin d'alléger la preuve.

(de l'ensemble $\{1, \ldots, j\}$) ; soit au contraire la décomposition utilise au moins $m+1$ mots de L_i, on marque alors tous ces mots à l'aide du marqueur $*$. On note M le langage correspondant aux mots de $\mathrm{NShuffle}(L_1 \cup \cdots \cup L_k)$ marqués comme on vient de l'expliquer.

> On peut construire un LCA qui devine ce marquage et reconnaît les mots du langage $\mathrm{NShuffle}(L_1 \cup \cdots \cup L_k)$.

On décrit un LCA qui reconnaît le langage M, on peut alors construire un LCA qui reconnaît le langage $\mathrm{NShuffle}(L_1 \cup \cdots \cup L_k)$ en utilisant le non déterminisme des automates afin de deviner le marquage.
Construction du LCA qui reconnaît le langage M.
On définit un LCA qui reconnaît le langage M. La partie finie de l'automate vérifie que pour chacun des $i \in \{1 \ldots k\}$ et $j \leq m$ le mot ne contient pas deux positions adjacentes étiquetées par $(A_i \cup B_i) \times \{j\}$. Un état de l'automate contient donc en particulier l'information de l'étiquette de la position courante. La contrainte linéaire demande à ce que, pour chaque $i = 1, \ldots, k$, une des propriétés suivantes soit vraie :

1. Il existe un $j \leq m$, tel qu'il y a exactement une occurrence des lettres de $A_i \times \{1, \ldots, j\}$ et aucune occurrence des lettres de $(A_i \cup B_i) \times \{j+1, \ldots, m, *\}$.
2. Pour toute paire de lettres $a, a' \in A_i$ le nombre de lettres $(a, *)$ et le nombre de lettres $(a', *)$ sont égaux, et strictement plus grand que m. Et il n'y a pas de lettre de $(A_i \cup B_i) \times \{1, \ldots, m\}$.

Cet automate et sa contrainte linéaire sont de taille $\mathcal{O}(m^2 \cdot k)$.

Par construction tout mot de M est accepté par un tel automate. Il reste donc à montrer la réciproque. Étant donné un mot w accepté par l'automate, on montre que ce mot appartient à M.

On note $Z \subseteq \{1, \ldots, k\}$ l'ensemble des marqueurs i tels qu'une lettre de $(A_i \cup B_i) \times \{*\}$ apparaît dans w. Par définition de l'automate, une lettre de $(A_i \cup B_i) \times \{1, \ldots, m\}$ peut apparaître dans le mot si et seulement si i n'est pas dans Z. On construit une décomposition π du mot w qui prouve son appartenance à M.

Une décomposition consiste à associer à chaque position du mot w une couleur telle que la projection du mot sur chaque couleur appartienne à un langage $LC(A_i, B_i)$ pour un certain i. On parle dans la suite de *i-couleur* en référence à ce i.

On dit qu'une décomposition est **propre** si :

1. Si $i \notin Z$ et $a, b \in A_i$, deux positions étiquetées respectivement par (a, j_1) et (b, j_2) sont associées à la même couleur si et seulement si on a $j_1 = j_2$.
2. Si $i \in Z$, toutes les lettres étiquetées $B_i \times \{*\}$ sont associées à au plus trois couleurs de la décomposition.

On montre par induction sur la longueur de w que pour tout $n \leq |w|$ il existe une décomposition propre ϕ telle que toutes les positions adjacentes situées avant la $n^{\text{ème}}$ position incluse sont associées à des couleurs différentes (c'est-à-dire une décomposition propre qui vérifie la condition de voisinage sur la première partie du mot de longueur n). Pour $n = |w|$, cette propriété correspond exactement à dire que w appartient au langage M.

Le cas $n = 0$ correspond à montrer l'existence d'une décomposition propre sur le mot w. Les contraintes lineaires de l'automate assurent la cohérence du nombre d'occurrences de chaque ensemble de type A_i. Les langages unitaires n'imposent aucune contrainte d'ordre entre les positions, on peut donc choisir les couleurs de la décomposition soit propre.

On suppose maintenant que π est une décomposition qui satisfait la propriété ci-dessus pour n. On construit une décomposition π' qui satisfait la propriété pour $n + 1$.

- Si l'une des positions n ou $n + 1$ du mot w est étiquetée par une lettre du type $(\_, j)$ pour $j, j' \in \{1, \ldots, m\}$, par définition de l'automate, l'autre position est étiquetée par $(\_, *)$ ou $(\_, j')$ avec $j \neq j'$. La décomposition π est propre donc les couleurs associées à ces deux positions sont différentes. On peut choisir $\pi' = \pi$.
- Si une des deux positions n ou $n+1$ est de la forme $(b, *)$ avec $b \in B_i$. On note l cette position ; il y a deux situations à considérer :
  - Soit la couleur associée aux positions n et $n+1$ par la décomposition π sont différentes. On choisit alors $\pi' = \pi$.
  - Soit π associe la même couleur aux deux positions. On construit une nouvelle décomposition π' qui est égale à π pour toutes les positions sauf la position l. La décomposition π étant propre, il existe trois couleurs[13] associées au positions $(b, *)$ dans π. Il existe donc une couleur χ parmi ces trois couleurs qui est différente des couleurs associées par π aux deux positions adjacentes à la position l. On choisit donc $\pi' = \pi$ pour toutes les positions différentes de la position l et $\pi'(l) = \chi$. Par construction π' est une décomposition propre qui vérifie la condition de voisinage jusqu'à la position $n + 1$.
- Si les positions n et $n + 1$ sont étiquetées par $(a, *)$ et $(a', *)$ pour $a \in A_i$ et $a' \in A_{i' \neq i}$, la décomposition associe forcément des couleurs différentes à ces deux positions car elles n'appartiennent pas au même langage. On peut choisir $\pi' = \pi$.
- Si les deux positions n et $n + 1$ sont étiquetées par $(a, *)$ et $(a', *)$ pour $a, a' \in A_i$:
  - Soit la décomposition π associe des couleurs différentes à ces deux positions. Dans ce cas on choisit $\pi = \pi'$.
  - Soit la décomposition π associe la même couleur χ aux deux positions. Supposons de plus qu'aucune lettre de $(b, *)$ pour $b \in B_i$ n'est associée à la couleur χ dans π[14] ; il existe donc exactement $|A_i|$ positions associées à χ dans π. Par definition du marquage et de la décomposition, il existe au moins m couleurs différentes associées aux positions $(a', *)$ dans π. Comme on a choisi $m > |A_i|$, il existe une position $(a', *)$ qui n'est ni

[13]S'il y a moins de trois couleurs, on complète ce triplet par des couleurs choisies arbitrairement parmi les couleurs associées aux lettres $(a, *)$ pour $a \in A_i$. Il en existe au moins $m \geq 3$.

[14]Si la couleur χ est associée à une lettre $(b, *)$, on choisit une couleur différente pour les $(b, *)$ associés à χ. On note ϕ, χ_b et ϕ'_b les couleurs associées aux $(b, *)$ dans π. Si une position $(b, *)$ colorée χ dans π n'est pas adjacente aux deux autres couleurs χ_b, $\chi_{b'}$, on choisit de lui associer une des deux couleurs χ_b, $\chi_{b'}$. Par cette transformation, les seules positions $(b, *)$ de la nouvelle décomposition qui sont associées à ϕ sont adjacentes à une position asociée à χ_b et une position associée à $\chi_{b'}$. Comme $m > 3$, il existe un couleur différente de χ_b, $\chi_{b'}$ avec laquelle on peut associer ces positions tout en gardant une décomposition propre qui vérifie l'hypothèse de récurence.

associée à χ ni adjacente à une position associée à χ dans π. On notre l' cette position et χ' la couleur associée. On choisit π' telle que $\pi'(n+1) = \chi'$, $\pi'(l') = \chi$ et $\pi' = \pi$ partout ailleurs. Par construction, π' est bien une décomposition propre qui satisfait les conditions de voisinage jusqu'à la position $n+1$.

On a ainsi construit une décomposition π' propre et qui vérifie les conditions de voisinage jusqu'à la $n+1^{\text{ème}}$ position de w.

Le mot w est donc bien dans le langage M.
Complexité.
L'automate à contrainte linéaire qui reconnaît le langage NShuffle$(LC(A_1, B_1) \cup \cdots \cup LC(A_k, B_k))$ est de taille au plus $\mathcal{O}(m^2 \cdot k)$ où m est la taille maximale des alphabets $A_i \cup B_i$. □

On fait maintenant le lien entre les shuffles marqués et les shuffles de voisinage.

Lemme 2.82 Étant donné un langage unitaire $LC(A, B)$, il existe un alphabet Γ, une substitution régulière telle que :

$$\text{Shuffle}_{\#}(LC(A,B)) = h(\text{NShuffle}(\bigcup_j L(\Gamma_j, \Delta)))$$

Le nombre de langages $L(\Gamma_j, \Delta)$ est exponentiel dans la taille de A. La taille des alphabets Γ_i et Δ est linéaire dans la taille de $A \cup B$.

Preuve. L'idée est de regrouper les parties du mot situées entre deux marqueurs # dans une lettre de l'alphabet Γ.

On appelle $\mathcal{P}_A$ l'ensemble des partitions de A. Pour toute partition σ de l'alphabet A en p ensembles, on note σ_i pour i entre 1 et p les ensembles de lettres correspondant. L'alphabet Γ contient les lettres γ^i_σ pour $i = 1, .., p$. L'alphabet Γ contient de plus une lettre spéciale appelée γ qui constitue l'unique lettre de l'alphabet Δ. Une partition σ définit un alphabet $\Gamma_j = \{\gamma^i_\sigma\}$. L'alphabet Γ est de taille exponentielle dans la taille de A.

La substitution régulière associe à la lettre γ le langage des mots de B^* qui commencent pas une position # et à toute lettre γ^i_σ les mots du langage[15] $LC(\sigma^i, B)$ qui commencent par #. La substitution régulière est donc aussi de taille exponentielle en $|A|$.

Le résultat est immédiat d'après la définition de h et des langages $L(\Gamma_i, \Delta)$. □

Ce résultat se généralise à tout shuffle marqué d'une union de langages unitaires.

Le Résultat

On a maintenant tous les éléments pour montrer la borne supérieure du Théorème 2.63

Proposition 2.83 Le problème de satisfaisabilité pour $\text{FO}^2(\sim, +1)$ est 3NExpTime.

[15]Ce langage est de taille exponentielle en $|\sigma^i|$ et polynomiale en $|B|$.

Preuve.
Forme normale. Étant donnée une formule ϕ de $\mathrm{FO}^2(\sim,+1)$, on peut construire une formule qui reconnaît les mêmes modèles et est de la forme :

$$\exists R_1 \ldots \exists R_\ell \bigvee \psi_i$$

où l est de taille exponentielle en la taille de ϕ , la disjonction est de taille doublement exponentielle et les formules ψ_i sont des conjonctions de taille exponentielle de formules simples de type (*reg*), (#), (*uniq*), $(\alpha|\beta)$ et $(\alpha \subseteq \beta)$, qui sont elles aussi de taille exponentielle dans la taille de ϕ.

De la forme normale aux langages unitaires. Étant donné une formule ψ en forme normale de donnée, la projection sur l'alphabet fini du langage correspondant, s'exprime comme $L \cap \mathrm{Shuffle}_\#(\bigcup_i L(A_i, B_i))$ où L est un langage régulier et $\mathrm{Shuffle}_\#$ correspond à l'opération Shuffle restreinte aux mots qui acceptent une décomposition marquée par le prédicat #. La taille de L ainsi que le nombre de langages $L(A_i, B_i)$ sont exponentiels dans la taille de ψ et $|A_i|$ est polynomiale dans la taille de ψ.

Des langages unitaires aux LCA. Un tel shuffle peut être reconnu par un automate LCA en utilisant une substitution régulière et l'opération NShuffle. Le Lemme 2.82 s'étend et on a la résultat suivant :

$$\mathrm{Shuffle}_\#(\bigcup_i LC(A_i, B_i)) = h(\mathrm{NShuffle}(\bigcup \{L_{p_i} : p_i \text{ partition de } A_i\}))$$

où les langages L_{p_i} sont des langages unitaires définis comme dans la preuve du Lemme 2.82. Les alphabets $|A_i|$ sont en nombre exponentiel et de taille linéaire dans la taille de ψ. Le shuffle de voisinage est donc constitué d'un nombre exponentiel de langages unitaires L_{p_i} chacun défini sur un alphabet de taille exponentielle dans la taille de ψ. La substitution h est de taille au plus doublement exponentielle dans la taille ψ.

Par les résultats des Lemmes 2.81, 2.79 et 2.77, on en déduit que le langage $L \cap \mathrm{Shuffle}_\#(\bigcup_i L(A_i, B_i))$ des mots reconnus par la formule ϕ est reconnu par un automate LCA de taille au plus doublement exponentielle dans la taille de ψ qui peut être construit en temps doublement exponentiel.

On revient à la réduction de la formule ϕ sous la forme $\exists R_1 \ldots \exists R_\ell \bigvee \psi_i$. La taille de chaque ψ_i est exponentielle dans la taille de la formule ϕ ; d'après ce qui précède, on peut construire en temps triplement exponentiel dans $|\phi|$ un automate LCA qui reconnaît le langage défini par ψ_i. Le nombre de ψ_i est doublement exponentiel dans la taille de ϕ. L'automate LCA qui reconnaît la projection sur l'alphabet fini du langage défini par ϕ est donc de taille triplement exponentielle dans la taille de ϕ. Le vide des LCA est NP d'après le Théorème 2.76.

La complexité de chacune des étapes de la transformation est au plus 3NExpTime dans la taille de ϕ.

Le problème de satisfaisabilité pour $\mathrm{FO}^2(\sim,+1)$ *est 3*NExpTime. $\square$

2.6 Extensions

2.6.1 Extensions décidables

La logique monadique existentielle EMSO.

Proposition 2.84 Le problème de satisfaisabilité pour $\mathrm{EMSO}^2(\sim, <, +1)$ est décidable.

La décidabilité du problème de satisfaisabilité des formules $\mathrm{EMSO}^2(\sim, <, +1)$ est une conséquence immédiate de la preuve du Théorème 2.52. En effet, les automates à données étant clos par renommage, la preuve de la Proposition 2.53 s'étend immédiatement aux formules de la logique $\mathrm{EMSO}^2(\sim, <, +1)$.

Sans valeur de données la logique $\mathrm{EMSO}^2(+1)$ permet de décrire n'importe quel langage régulier alors que la logique $\mathrm{FO}(<)$ représente la classe restreinte des langages sans-étoile[16]. Avec données la logique $\mathrm{EMSO}^2(\sim, <, +1)$ est strictement plus expressive que la logique $\mathrm{FO}^2(\sim, <, +1)$. En particulier cette dernière ne peut pas décrire le langage constitué de l'ensemble de tous les mots de données dont la projection sur l'alphabet fini appartient au langage $(aa)^*$ qui est régulier mais n'est pas sans-étoile.

Si on s'intéresse à l'expressivité des automates à données, on en déduit que les automates à données sont strictements plus expressifs que la logique $\mathrm{FO}^2(\sim, <, +1)$ puisqu'ils peuvent décrire $\mathrm{EMSO}^2(\sim, <, +1)$. Cependant la logique $\mathrm{EMSO}^2(\sim, <, +1)$ ne permet pas de décrire tous les langages définissables par les automates à données. Cela vient du fait qu'un automate à données peut, grâce à l'automate de classe, comparer les positions successives d'une même classe.

Proposition 2.85 La logique $\mathrm{EMSO}^2(\sim, <, +1)$ est strictement moins expressive que les automates de données.

Exemple 2.86 On note $L_{(aa)^*}$ l'ensemble des mots de données sur un alphabet à une lettre tels que toutes les classes sont de longueur paire.

On peut facilement reconnaître $L_{(aa)^*}$ à l'aide d'un automate de données : l'automate de base recopie le mot et l'automate de classe reconnaît $(aa)^*$. Cependant ce langage ne peut pas être décrit par une formule de la logique $\mathrm{EMSO}^2(\sim, <, +1)$.

Lemme 2.87 Le langage $L_{(aa)^*}$ ne peut être défini par aucune formule de la logique $\mathrm{EMSO}^2(\sim, <, +1)$.

Preuve. Grâce à la transformation des formules $\mathrm{FO}^2(\sim, <, +1)$ en forme normale de Scott donnée dans le Lemme 2.54, on peut se restreindre aux formules $\mathrm{EMSO}^2(\sim, <, +1)$ de la forme $\exists R_1 \ldots \exists R_k\ \forall x \forall y\ \chi_0 \wedge \bigwedge_i \forall x \exists y\ \chi_i$ où les formules χ sont des formules $\mathrm{FO}^2(\sim, <, +1)$ sans quantificateur sur l'alphabet étendu par les k prédicats unaires $R_1, \ldots, R_k$.

On raisonne sur le nombre k de prédicats unaires introduits par la quantification existentielle du second ordre et on montre que :

[16] On rappelle que $\mathrm{FO}(<, +1)$ et $\mathrm{FO}(<)$ sont équivalentes du moment qu'on autorise l'utilisation de trois variables.

> Pour tout k il existe un mot $u \in L_{(aa)^*}$ tel que, pour toute valuation des prédicats $R_1, \ldots R_k$ il existe un mot $v \notin L_{(aa)^*}$ et une valuation des prédicats sur v tels que les deux mots munis de leur valuation sont indistingables par des formules de la forme $\forall x \forall y\ \chi(x,y)$ ou $\forall x \exists y\ \chi(x,y)$ où χ est une formule $\mathrm{FO}^2(\sim,<,+1)$ sans quantificateur sur l'alphabet étendu par les k prédicats unaires.

Cela implique que le langage $L_{(aa)^*}$ n'est pas définissable par une formule dont la quantification existentielle est de taille k.

Une valuation de k prédicats sur un mot u peut être vue comme une fonction de coloriage de l'ensemble des positions du mot dans un ensemble de $m = 2^k$ couleurs[17]. Dans la suite, on parlera de coloriage et de couleur dans ce sens. L'alphabet Σ contient une seule lettre. Pour plus de clarté, on omet dans la suite l'étiquetage des mots que l'on donne comme une suite de valeurs de données de $\mathcal{D} = \mathbb{N}$.

On fixe un entier k et on note $m = 2^k$ le nombre de couleurs correspondant. On choisit le mot $u = (12\ldots n)^M$ avec $n = 3(m^3+1)$ et $M = 4.m^n.2^{4mn}$. On considère π un coloriage quelconque du mot u.

Étant donnée une position x dans le mot, on note $C_x^{\sim,\lhd}$ l'ensemble des couleurs de π qui apparaissent à gauche de la position x dans des positions de la même classe que x. On note $C_x^{\nsim,\lhd}$ l'ensemble des couleurs de π qui apparaissent à gauche de la position x dans des classes différentes de celle de x. De même, on définit $C_x^{\sim,\rhd}$, $C_x^{\nsim,\rhd}$ comme les ensembles de couleurs apparaissant à droite de la position x respectivement dans la classe de x ou dans une classe différente.

Puisque le mot u contient $4.(m.2^{4m})^n$ copies[18] du bloc $12\ldots n$, il existe quatre blocs coloriés de la même manière par π et qui vérifient la propriété suivante : si x et y sont deux positions de ces blocs appartenant à la même classe alors les ensembles $C_x^{\sim,\lhd}$, $C_x^{\nsim,\lhd}$, $C_x^{\sim,\rhd}$ et $C_x^{\nsim,\rhd}$ et $C_y^{\sim,\lhd}$, $C_y^{\nsim,\lhd}$, $C_y^{\sim,\rhd}$ et $C_y^{\nsim,\rhd}$ sont les mêmes. Dans la suite, on désigne par *bloc* A, *bloc* B les deux blocs centraux d'un quadruplet de blocs vérifiant la propriété ci-dessus. Un bloc est de longueur $3(m^3+1)$, il existe donc dans le bloc B deux positions α et β non adjacentes coloriées de la même couleur et dont les positions prédécesseures et successeures sont ausi coloriées de la même façon[19]. Dans la suite on appelle i et j les classes des positions α et β.

On construit v à partir de u en modifiant la classe la position β du bloc B ; dans le mot v cette position est dans la classe i. On garde le coloriage π pour colorier v.

Par construction, u est un mot du langage $L_{(aa)^*}$ (chaque classe possède M éléments et M est pair) alors que v n'appartient pas au langage (les classes i et j sont de taille impaire).

Il reste maintenant à montrer que l'on ne peut pas distinguer les mots u et v par des formules du type $\forall x \forall y\ \chi(x,y)$ ou $\forall x \exists y\ \chi(x,y)$ où χ est une formule $\mathrm{FO}^2(\sim,<,+1)$ sans quantificateur.

[17] La couleur d'une position décrit la valuation des k prédicats dans cette position.

[18] $m.2^{4m}$ correspond au nombre de solutions possibles dans une position x pour la couleur $\pi(x)$ et les ensembles $C_x^{\sim,\lhd}$, $C_x^{\nsim,\lhd}$, $C_x^{\sim,\rhd}$ et $C_x^{\nsim,\rhd}$.

[19] On remarque que les positions α et β du bloc A vérifient les mêmes propriétés.

Les formules $\forall\forall$ *:* On montre que pour toute paire de positions du mot u, il existe une paire de positions du mot v coloriées de la même manière par π qui vérifient les mêmes formules atomiques.
Si aucune des deux positions du mot u n'est dans le bloc B, on peut choisir la même paire dans le mot v. Sinon, il suffit de choisir à la place de chaque position du bloc B une position correspondante dans un des blocs de même coloriage. Puisqu'il existe un bloc $1 \dots m$ à gauche et un bloc $1 \dots m$ à droite du bloc B qui sont coloriés de la même manière que le bloc B par π, c'est toujours possible.

De la même manière, pour toute paire de positions de v, il existe une paire de positions dans u coloriées de la même manière et qui vérifient les mêmes formules atomiques.

Cela garantit que l'on ne peut pas différencier les deux mots coloriés à l'aide d'une formule de la forme $\forall x \forall y\ \chi(x, y)$ où χ est sans quantificateur.

Les formules $\forall\exists$ *:* On montre d'abord que pour toute position x du mot u, il existe une position x' dans v telle que pour toute position y dans u (resp. y' dans v), on peut trouver une position y' dans v (resp y dans u) telle que les paires (x, y) et (x', y') sont coloriées de la même manière par π et vérifient les mêmes formules atomiques.
Si x n'est pas une position du bloc B, on choisit la même position x' dans v. Si x est une position du bloc B, on choisit la position correspondante du bloc A dans v. On utilise le fait que pour chaque classe i, les ensembles $C^{\sim,\triangleleft}$, $C^{\not\sim,\triangleleft}$, $C^{\sim,\triangleright}$ et $C^{\not\sim,\triangleright}$ sont identiques pour les positions de la classe i dans les blocs A et B. Cela garantit que quelque soit la position y ou y' que l'on considère, on pourra trouver une position dans l'autre mot qui vérifie les bonnes propriétés.

De la même manière, on montre maintenant que pour toute position x' du mot v, il existe une position x dans u telle que pour toute position y dans u (resp. y' dans v), on peut trouver une position y' dans v (resp y dans u) telle que les paires (x, y) et (x', y') sont coloriées de la même manière par π et vérifient les mêmes formules atomiques.
Ce cas est symétrique du précédent, la seule différence est le cas où x' correspond à la position β, on choisit alors pour x la position α dans u. Par construction, ces deux positions ainsi que leurs successeurs et prédécesseurs ont la même couleur et on peut voir que les ensembles $C^{\sim,\triangleleft}$, $C^{\not\sim,\triangleleft}$, $C^{\sim,\triangleright}$ et $C^{\not\sim,\triangleright}$ sont aussi les mêmes entre les deux positions.

Cela garantit que l'on ne peut pas différencier les mots u et v à l'aide d'une formule de la forme $\forall x \exists y\ \chi(x, y)$ où χ est sans quantificateur.

Bilan : Étant donné un entier k on définit un mot $u \in L_{(aa)^*}$ tel que pour tout coloriage π de ce mot à l'aide de 2^k couleurs (correspondant aux 2^k valuations possibles d'un$k$-uplet de prédicats unaires), on peut définir un mot $v \notin L_{(aa)^*}$ de même longueur tels que u et v munis du même coloriage π sont indistingables par des formules de la forme $\forall x \forall y\ \chi(x, y)$ ou $\forall x \exists y\ \chi(x, y)$ où χ est une formule $\mathrm{FO}^2(\sim, <, +1)$ sans quantificateur sur l'alphabet étendu par les k prédicats unaires. Le langage $L_{(aa)^*}$ n'est donc pas définissable par une formule de $\mathrm{EMSO}^2(\sim, <, +1)$ dont la quantification existentielle est de taille k.

La construction étant valable pour tout entier k, le langage $L_{(aa)^*}$ n'est pas définissable par une formule de $\mathrm{EMSO}^2(\sim, <, +1)$. $\square$

Successeur de classe

L'expressivité des formules $\mathrm{EMSO}^2(\sim,<,+1)$ ne suffit pas à simuler complètement les automates à données. En effet un automate à données peut, grâce à l'automate de classe, comparer les positions successives d'une même classe. On introduit ici un nouveau prédicat binaire : **le successeur de classe** que l'on note $\oplus 1$. La formule atomique $y = x \oplus 1$ est vraie si et seulement x et y sont deux positions successives (y avant x) d'une même classe dans le modèle.

Comme pour l'extension quantificateurs existentiels du second ordre, la preuve de la Proposition 2.53 s'étend facilement aux formules de la logique $\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$. Il en découle le résultat suivant :

Proposition 2.88 Étant donnée une formule de $\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$, on peut construire un automate à données de taille doublement exponentielle qui reconnaît le même langage.

Réciproquement, on peut décrire le langage d'un automate à données à l'aide d'une formule de la logique étendue.

Proposition 2.89 Étant donné un automate à données, on peut construire une formule $\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$ de taille polynomiale qui reconnaît exactement les mots de données acceptés par cet automate.

Preuve. Étant donné un automate à données $(\mathcal{A},\mathcal{B})$, on construit une formule $\phi_{(\mathcal{A},\mathcal{B})}$ de $\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$ telle qu'un mot de données est accepté par l'automate à données si et seulement s'il vérifie la formule. Cette construction est très proche de la construction classique d'une formule $\mathrm{EMSO}^2(+1)$ (ou plus généralement MSO) pour décrire un langage régulier reconnu par un automate fini.

L'idée est d'utiliser les prédicats du second ordre pour ajouter au mot de données les informations correspondant à une exécution de l'automate à données. On associe à une exécution une valuation des prédicats du second ordre. Cette valuation permet de marquer dans chaque position du mot : (1) l'état de $\mathcal{A}$ et la lettre de l'aphabet Γ correspondant à l'exécution de l'automate $\mathcal{A}$ à cette position, (2) l'état de l'automate $\mathcal{B}$ correspondant à cette position dans l'exécution de $\mathcal{B}$ sur la classe correspondante. On peut à l'aide d'une formule assurer qu'un tel marquage d'un mot correspond bien à une exécution acceptante de l'automate à données.

La formule est de la forme suivante :

$$\exists R_1 \ldots R_k \ \phi_R \wedge \phi_{\mathcal{A}} \wedge \phi_{\mathcal{B}}$$

- Les prédicats du second ordre décrivent l'ensemble des états de $\mathcal{A}$, l'alphabet Γ de sortie du transducteur ainsi que l'ensemble des états de $\mathcal{B}$.
- La formule ϕ_R assure que chaque position du mot contient exactement un état de $\mathcal{A}$, une lettre de Γ et un état de $\mathcal{B}$.
- La formule $\phi_{\mathcal{A}}$ vérifie que le coloriage par les états de $\mathcal{A}$ et Γ des positions du mot correspond bien à une exécution acceptante de l'automate $\mathcal{A}$.
- La formule $\phi_{\mathcal{B}}$ vérifie que le coloriage par Γ et les états de $\mathcal{B}$ correspond bien pour chaque classe à une exécution acceptante de l'automate $\mathcal{B}$.

Le nouveau prédicat $\oplus 1$ est utilisé (et nécessaire) uniquement dans la formule ϕ_B.

On note $Q_{\mathcal{A}}$ et $Q_{\mathcal{B}}$ l'ensemble des états des automates $\mathcal{A}$ et $\mathcal{B}$. Modulo renommage, on peut supposer que ces deux ensembles sont disjoints. L'ensemble des prédicats $R_1, \ldots, R_k$ est constitué d'un prédicat R_q pour chaque état q de $Q_{\mathcal{A}} \cup Q_{\mathcal{B}}$ et d'un prédicat R_α pour chaque lettre α de Γ.

$$\begin{array}{lll}\phi_R ::= \forall x & & \bigvee_{q \in Q_{\mathcal{A}}} R_q(x) \\ & \wedge & \bigwedge_{q,q' \in Q_A,\ q \neq q'} \neg[R_q(x) \wedge R_{q'}(x)] \\ & \wedge & \bigvee_{q \in Q_{\mathcal{B}}} R_q(x) \\ & \wedge & \bigwedge_{q,q' \in Q_B,\ q \neq q'} \neg[R_q(x) \wedge R_{q'}(x)] \\ & \wedge & \bigvee_{\alpha \in \Gamma} R_\alpha(x) \\ & \wedge & \bigwedge_{\alpha,\alpha' \in \Gamma,\ \alpha \neq \alpha'} \neg[R_\alpha(x) \wedge R_{\alpha'}(x)]\end{array}$$

On décrit maintenant la formule $\phi_{\mathcal{A}}$. Dans la conjonction, q et q' représentent des états de $\mathcal{A}$, q_0 l'état initial, $Q^f_{\mathcal{A}}$ l'ensemble des étas finaux de $\mathcal{A}$, a une lettre de l'aphabet d'entrée Σ, α une lettre de l'aphabet de sortie Γ et $\delta_{\mathcal{A}}$ la fonction de transition de l'automate $\mathcal{A}$.

$$\begin{array}{lll}\phi_{\mathcal{A}} ::= & \forall x & \bigwedge_{(q,a,\alpha,q') \notin \delta_{\mathcal{A}}} \neg\left[R_q(x) \wedge a(x) \wedge R_\alpha(x) \wedge \exists y[y = x+1 \wedge R_{q'}(y)]\right] \\ \wedge & \exists x & debut(x) \wedge R_{q_0}(x) \\ \wedge & \exists x & fin(x) \wedge \left[\bigvee_{q \in Q^f_{\mathcal{A}}} R_q(x)\right]\end{array}$$

Enfin la formule ϕ_B vérifie la partie correspondant à l'automate de classe. Dans cette formule, q et q' représentent des états de $\mathcal{B}$, q_I l'état initial de $\mathcal{B}$, $Q^f_{\mathcal{B}}$ l'ensemble des étas finaux de $\mathcal{B}$, α une lettre de l'alphabet Γ et $\delta_{\mathcal{B}}$ la fonction de transition de l'automate $\mathcal{B}$.

$$\begin{array}{lll}\phi_{\mathcal{B}} ::= & \forall x & \bigwedge_{(q,\alpha,q') \notin \delta_{\mathcal{B}}} \neg\left[R_q(x) \wedge R_\alpha(x) \wedge \exists y[y = x \oplus 1 \wedge R_{q'}(y)]\right] \\ \wedge & \forall x & \mathrm{first}(x) \rightarrow R_{q_I}(x) \\ \wedge & \forall x & \mathrm{last}(x) \rightarrow \left[\bigvee_{q \in Q^f_{\mathcal{B}}} R_q(x)\right]\end{array}$$

Par construction la formule représente bien l'ensemble des mots acceptés par l'automate à données.

On peut encoder la même information avec un nombre logarithmique de prédicats (n prédicats permettent de représenter un alphabet de 2^n lettres). Cette représentation est plus compacte mais aussi moins claire à présenter. La formule se construit de la même façon. □

On déduit des deux propositions précédentes les deux résultats suivants :

Théorème 2.90 *Les automates à données et la logique* EMSO^2($\sim$,+1,$\oplus$1) *ont le même pouvoir d'expression et les traductions sont effectives.*

Proposition 2.91 La satisfaisabilité de EMSO^2($\sim$,+1,$\oplus$1) est décidable.

Successeurs multiples

La restriction à deux variables des formules considérées ne semble pas permettre de comparer les valeurs de données de trois positions consécutives d'un mot. Une telle opération est naturelle et ne semble pas ajouter trop de pouvoir expressif à nos formules. Le but est de pouvoir exprimer des motifs locaux dans le mot. Pour cela, on considère la logique FO^2 obtenue en ajoutant les formules atomiques $y = x + k$ pour $k \in \mathbb{N}$. On note cette logique FO$^2(\sim, <, +\omega)$. Bien entendu, une formule donnée utilise un nombre fini de ces nouvelles formules atomiques. Le problème de satisfaisabilité pour cette logique reste décidable.

Théorème 2.92 *Le problème de satisfaisabilité de* $\mathrm{FO}^2(\sim, <, +\omega)$ *est décidable.*

Preuve. Ce résultat s'obtient en adaptant le preuve du Théorème 2.52, on explique les différences entre les deux preuves.

La preuve du Théorème 2.52 définit et utilise le prédicat S_{+1}. Dans le cas de la logique $\mathrm{FO}^2(\sim, <, +\omega)$, on définit de manière similaire les prédicats S_{+j}, pour tout entier $j \geq 1$: le prédicat S_{+j} est vrai à la position x si et seulement si $x \sim (x + j)$. Dans l'Exemple 2.42, on montre comment construire un automate à registres $\mathcal{A}_+$ qui vérifie que les positions vérifiant S_{+1} sont exactement celles qui sont dans la même classe que leur successeur. En utilisant la même idée, il est facile de construire un automate à registres A_{+j} qui vérifie que le prédicat S_{+j} vérifie la propriété voulue. Par la Proposition 2.40 pour tout automate à registres on peut construire un automate à données qui reconnaît le même langage. On peut donc construire un automate à données D_{+j} qui devine et vérifie les positions qui satisfont le prédicat S_{+j}.

On considère une formule ϕ de $\mathrm{FO}^2(\sim, <, +\omega)$. Cette formule utilise un nombre fini de prédicats $+j$ appartenant à un ensemble $\{+1, \ldots, +k\}$ avec $k \in \mathbb{N}$. Comme dans le cas FO^2, on commence par mettre φ en forme normale de Scott. Il faut maintenant expliquer comment s'étendent les Lemmes 2.56 et 2.57.

Si on regarde les formules de la forme (2.1). Comme dans le Lemme 2.56, le cas non-trivial est le cas où $\delta(x, y) = (x \not\sim y)$. Les prédicats $+j$ entraînent une multiplication des contraintes de localisation entre x et y. Cependant, on peut traiter ce cas de la même manière que dans le Lemme 2.57.

En ce qui concerne les formules de la forme (2.2) traitées dans le Lemme 2.57, la seule différence apparaît dans les sous-formules $\epsilon(x, y)$ qui sont maintenant de la forme suivante : $x = y \pm j$ ou $x < y \pm j$, $0 \leq j \leq k$. La preuve s'étend naturellement. Par exemple, si $\delta(x, y) = (x \not\sim y)$ on marque les deux classes correspondant respectivement aux positions Lst_β et Llst_β pour traiter le cas $\epsilon(x, y) = (x + j < y)$, et on utilise le prédicat S_{+j} pour traiter le cas où $\epsilon(x, y) = (x + j = y)$. □

La preuve du Théorème 2.92 montre que la logique $\mathrm{FO}^2(\sim, <, +\omega)$ est incluse dans la logique $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$, ce qui n'est pas évident a priori. De manière plus précise, on a le résultat suivant :

Proposition 2.93 La logique $\mathrm{EMSO}^2(\sim, <, +\omega)$ est strictement moins expressive que la logique $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$.

Preuve (Idées). Le fait que la logique $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$ soit au moins aussi expressive que $\mathrm{EMSO}^2(\sim, <, +\omega)$ découle de la preuve du Théorème 2.92 et du Théorème 2.90. La preuve du Théorème 2.92 montre que tout langage définissable par une formule $\mathrm{EMSO}^2(\sim, <, +\omega)$ est reconnu par un automate à données. Le Théorème 2.90 dit que les automates à données et la logique $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$ ont la même expressivité.

Pour montrer que l'inclusion est strite, on peut utiliser le langage $L_{(aa)^*}$ de l'Exemple 2.86 qui correspond aux mots de données dont toutes les classes sont de longueur paire. Ce langage est définissable par un automate à données, donc par une formule $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$. D'après le Lemme 2.87, ce langage n'est pas définissable par une formule $\mathrm{EMSO}^2(\sim, <, +1)$. On peut montrer que ce langage n'est pas définissable par une formule de $\mathrm{EMSO}^2(\sim, <, +\omega)$. L'idée

est la suivante : une formule de $\mathrm{EMSO}^2(\sim, <, +\omega)$ utilise un nombre fini de prédicats $+k$ pour $k \in \mathbb{N}$. Or la distance entre une position et son successeur de classe dans les mots du langage L_{aa^*} n'est pas bornée.
La preuve du Lemme 2.87 s'étend au cas des formules $\mathrm{EMSO}^2(\sim, <, +\omega)$. La difficulté est d'ordre combinatoire, en plus du nombre de prédicats unaires introduits par quantification existentielle, on doit aussi raisonner sur le plus grand entier n tel que le prédicat $+n$ ou $-n$ apparaît dans une formule donnée.

$\square$

Les mots infinis

On a présenté jusqu'ici des extensions de la logique. Une autre extension naturelle est d'étendre les modèles considérés. On s'intéresse ici au cas des mots infinis de données.

Théorème 2.94 *Le problème de satisfaisabilité de* $\mathrm{FO}^2(\sim, <, +1)$ *sur les mots de données infinis est décidable.*

La structure de la preuve de ce théorème est la même que celle du Théorème 2.52. On définit un modèle d'automate à données pour les mots infinis (ω-automate à données) et on montre qu'à partir d'une formule de FO^2 on peut construire un automate qui reconnaît le même langage. Le vide des ω-automates à données est décidable. Pour montrer ce résultat, on utilise une extension des automates multicompteurs au cas de mots infinis

Définition 2.95 Un **ω-automate à données** est un triplet $(\mathcal{B}, \mathcal{C}_f, \mathcal{C}_i)$ où :

1. l'**automate de base** $\mathcal{B}$ est un transducteur de Büchi lettre à lettre (Γ étant son alphabet de sortie) ;
2. l'**automate de classe fini** $\mathcal{C}_f$ est automate fini sur l'aphabet Γ ;
3. l'**automate de classe infini** $\mathcal{C}_i$ est automate de Büchi sur l'alphabet Γ.

Un mot de données infini w est accepté par l'automate à données si et seulement si (1) il existe une exécution acceptante de l'automate de base sur la projection sur l'alphabet fini de w qui renvoie le mot $b_1 b_2 \ldots$ tel que pour toute classe finie $i_1 < \cdots < i_k$ de w, le mot $b_{i_1} \ldots b_{i_k}$ est accepté par l'automate de classe fini $\mathcal{C}_f$; et (2) pour toute classe infinie $i_1 < i_2 < \ldots$ de w, le mot infini $b_{i_1} b_{i_2} \ldots$ est accepté par l'automate de classe infini $\mathcal{C}_i$.

Le Théorème 2.94 découle directement des Propositions 2.96, 2.99 et 2.101.

Proposition 2.96 Tout ω-langage de données définissable dans $\mathrm{FO}^2(\sim, <, +1)$ est reconnu par un ω-automate à données.

Preuve. Ce résultat se prouve de la même manière que celui de la Proposition 2.53. On explicite seulement les différences qui apparaissent dans les Lemmes 2.56 et 2.57.

On considère ψ une formule de type 2.1.

$$\forall x \forall y \left[\alpha(x) \ \wedge \ \beta(y) \ \wedge \ \delta(x,y)) \ \rightarrow \quad \gamma(x,y)\right]$$

Le cas où $\delta(x, y)$ est $x \sim y$ se traite de la même manière que dans le cas fini en utilisant le prédicat S_{+1}. On exprime la formule ψ comme des conditions régulières sur les classes (finies et infinies) que l'on peut alors vérifier à l'aide des deux automates de classe $\mathcal{C}_f$ and $\mathcal{C}_i$.

Si maintenant $\delta(x,y)$ est $x \not\sim y$, on étudie les différents cas comme dans la preuve du Lemme 2.56.

- Le cas $\gamma(x,y) = false$ est similaire au même cas pour les mots finis.
- Si $\gamma(x,y)$ est $x < y$, alors ψ est vraie si et seulement si (a) β n'apparaît pas dans le mot ou (b) α apparaît infiniment souvent dans au plus une classe. On peut vérifier cela à l'aide d'un automate à données qui devine dans quel cas on se trouve et vérifie soit (a) à l'aide de automate de base soit (b) en marquant la classe concernée puis en vérifiant que le marquage est correct et que α n'apparaît pas infiniment souvent dans une classe non marquée et infiniment souvent dans la classe marquée.
- Les cas où $\gamma(x,y)$ est $x = y+1$, $x \neq y = 1$ ou $x = y+1 \vee y = x+1$ se traitent de la même manière que dans le Lemme 2.56.

On considère maintenant les formules de type 2.2.

$$\forall x \exists y \quad \alpha(x) \rightarrow \ \big[\beta(y) \ \wedge \ \delta(x,y) \ \wedge \ \epsilon(x,y)\big]$$

Si $\delta(x,y)$ est $x \sim y$, on utilise les mêmes techniques que dans le Lemme 2.57, en vérifiant des propriétés régulières sur chaque classe en utilisant si besoin est le prédicat S_{+1}.

Si $\delta(x,y)$ est $x \not\sim y$, on sait que $x \neq y$:

- Si $\epsilon(x,y)$ est $x+1 = y$ ou $x = y+1$, on résout ces cas de la manière précédente en utilisant le prédicat S_{+1}.
- Si $\epsilon(x,y)$ est $x+1 < y$, alors ψ impose que pour chaque position α, il existe une position non adjacente à sa droite qui vérifie β et appartient à une classe différente.
  L'automate de base devine si (a) il y a un nombre fini de β dans le mot, (b) il existe une seule classe contenant un nombre infini de β, ou (c) il y a au moins deux classes contenant un nombre infini de β.
  La situation (a) est identique au cas fini. Dans la situation (b), l'automate marque la classe contenant un nombre infini de β et vérifie que le dernier α dans c se situe au plus de deux positions avant le dernier β qui n'est pas dans cette classe. Dans la situation (c), il suffit de marquer les deux classes et de vérifier qu'elles contiennent chacune une infinité de β.
- Si $\epsilon(x,y)$ est $y+1 < x$, il faut vérifier que (a) il n'y a pas de α avant la position $\mathrm{Fst}_\beta + 2$ et (b) tous les α situés entre $\mathrm{Fst}_\beta + 2$ et $\mathrm{Ffst}_\beta + 2$ sont dans une classe différente de $\mathrm{Fst}_\beta(w)$. On étend pour cela les notations introduites dans la Section 2.2 aux mots de données infinis. On remarque que (a) et (b) peuvent aussi être vérifiés si Fst_β et Ffst_β ne sont pas définis.

$\square$

On définit le modèle des "automates à sacs de Büchi". Ces automates ont essentiellement des automates multicompteurs étendus au cas des mots infinis. La condition d'acceptance ne dépend pas des états de l'automate comme dans un automate Büchi traditionnel mais du contenu des sacs.

Définition 2.97 Un **automate à sacs de Büchi**[20] est un automate fini muni d'un nombre fini de sacs qui peuvent contenir des jetons. Contrairement aux valeurs d'un compteur, on suppose que les jetons proviennent d'un ensemble infini et sont deux à deux distincts. L'automate peut effectuer des opérations sur les sacs :

- créer un nouveau jeton et le placer dans le sac i.
- déplacer un jeton du sac i dans le sac j.

Comme dans le cas des automates multicompteurs sur les mots finis, une transition dépend de la lettre courante et peut changer d'état et effectuer des opérations sur les sacs. Lorsqu'une transition déplace un jeton du sac i, celui-ci est choisi aléatoirement par la machine. La condition d'acceptation dépend d'un ensemble de sacs $\mathcal{F}$. Une exécution est acceptante si chaque jeton créé au cours de l'exécution apparaît dans un des sacs de $\mathcal{F}$ infiniment souvent. On remaque que l'identité des jetons n'est utilisée que pour la condition d'acceptation infinie.

Remarque 2.98 Ce modèle d'automate est très proche de celui des automates multicompteurs. Un compteur peut être vu comme un sac. Une incrémentation d'un compteur correspond à la création d'un jeton dans le sac correspondant. La décrémentation d'un compteur correspond à déplacer un jeton du sac correspondant au compteur vers un sac "poubelle". On pourrait ainsi définir un modèle d'automates à sacs pour les mots finis, une exécution serait acceptante si tous les jetons créés au cours de l'exécution sont placés dans les sacs acceptants à la fin de l'exécution. Ce modèle est équivalent aux automates multicompteurs dans le cas des mots finis.

Proposition 2.99 A partir d'un ω-automate à données $\mathcal{A}$, on peut construire un automate à sacs de Büchi qui reconnaît la projection sur l'alphabet fini du langage reconnu par $\mathcal{A}$.

Preuve. Étant donné un ω-automate à données $\mathcal{A} = (\mathcal{B}, \mathcal{C}_f, \mathcal{C}_i)$, on peut simuler son comportement à l'aide d'un automate à sacs de Büchi. L'automate à sacs reconnaît la projection sur l'alphabet fini des mots du langage $L_\mathcal{A}$ décrit par $\mathcal{A}$ et utilise des jetons afin de représenter les valeurs de données. L'automate de base $\mathcal{B}$ est directement représenté dans l'automate à sacs sans utiliser de jeton. On note S l'ensemble de ses états et $\Delta \subseteq S \times \Sigma \times \Gamma \times S$ l'ensemble de ses transitions. Les automates de classe $\mathcal{C}_f, \mathcal{C}_i$ sont quant à eux simulés à l'aide des sacs et des jetons. On note Q l'ensemble des états des automates de classes[21] et δ la relation de transition. Les états de l'automate à sacs sont des paires (s, γ) où $s \in S$ et $\gamma \in \Gamma$. Comme dans la preuve de la Proposition 2.49, on va deviner à chaque position la lettre γ engendrée par l'automate de base. Les états initiaux de l'automate à sacs sont les états (s, a) tels que s est initial pour l'automate de base et a est une lettre quelconque de Γ. L'automate à sacs a un sac par état $q \in Q$, un sac par état $s \in S$, et un sac poubelle. L'ensemble des sacs acceptants est constitué des sacs qui correspondent à des états acceptants de $\mathcal{B}$ ou $\mathcal{C}_i$, et du sac poubelle.

Étant donné un mot qui représente la projection sur l'alphabet fini d'un mot w de $L_\mathcal{A}$, l'idée est d'utiliser un jeton qui va se déplacer dans les sacs $s \in S$ pour tracer une exécution de $\mathcal{B}$ sur w et ainsi pouvoir tester la condition

[20]Cet automate reconnaît des mots de Σ^ω (sans valeur de données).

[21]On suppose que $\mathcal{C}_f, \mathcal{C}_i$ ont des ensembles d'états disjoints.

d'acceptation. De plus on va utiliser un jeton pour représenter chaque classe ; au cours de l'exécution, le sac $q \in Q$ contient l'ensemble des classes pour lesquelles l'automate de classe est dans l'état q.

Lorsque l'automate à sacs est dans l'état (s, γ) et la position courante est a une transition se décrit de la manière suivante : (0) l'automate passe dans un état (s', γ') tel que $(s, a, \gamma, s') \in \Delta$ et déplace un jeton du sac s au sac s'. Lors de la première transition l'automate crée un jeton dans le sac s' car tous les sacs sont vides. En ce qui concerne les sacs $q \in Q$, l'automate peut effectuer deux types de transitions (1) créer un nouveau jeton dans le sac q tel que $(q_0, \gamma, q) \in \delta$ (où q_0 représente un état initial de Q), ou (2) pour une transition $(p, \gamma, q) \in \delta$, l'automate à sacs déplace un jeton du sac p au sac q ou de manière non-déterministe dans le sac poubelle si q est un état final de $\mathcal{C}_f$. (3) Dans tous les cas, l'automate déplace tous les jetons présents dans le sac poubelle pour les remettre dans le sac poubelle. On appelle cette dernière opération une opération de “mélange”, elle est importance pour la condition d'acceptation.

On peut remarquer qu'au cours d'une exécution de l'automate à sacs, il y a exactement un seul jeton qui est déplacé dans l'ensemble des sacs $s \in S$ correspondant aux états de $\mathcal{B}$.

Étant donné un mot de données $w \in L_{\mathcal{A}}$, une exécution acceptante de $\mathcal{A}$ décrit l'exécution acceptante de l'automate à sacs. L'étape (0) des transitions simule l'exécution de $\mathcal{B}$. L'évolution des sacs q correspondant aux états Q simule les exécutions des automates de classes. Une transition choisit l'étape (1) si la position courante est la première de sa classe dans le mot w ; l'étape (2) si l'exécution de l'automate de classe sur la classe correspondant à la position courante est dans l'état p avant celle-ci. Comme l'exécution simulée de $\mathcal{B}$ est acceptante, le jeton qui se déplace dans les sacs $s \in S$ passe infiniment par un sac correspondant à un état acceptant de $\mathcal{B}$ qui est un sac acceptant pour l'automate à sacs. De la même manière les jetons créés dans des sacs q correspondant à des états de $\mathcal{C}_i$ passent infiniment souvent dans un sac acceptant de l'automate car ils correspondent chacun à une exécution acceptante de $\mathcal{C}_i$. L'étape (3) de mélange sert à terminer les exécutions de l'automate de classe $\mathcal{C}_f$. Une fois que toutes les positions correspondant à une classe finie ont été visitées, on ne peut pas détruire le jeton correspondant. Si l'execution de $\mathcal{C}_i$ sur une classe est acceptante, le jeton correspondant à cette classe va finir dans l'état poubelle afin qu'il ne cause pas l'échec de l'exécution de l'automate à sacs. Il visitera ainsi infiniment souvent un état acceptant.

Réciproquement, supposons qu'un w mot est accepté par l'automate à sacs. On considère une exécution acceptante de l'automate à sacs. À chaque transition il y a toujours exactement un jeton qui est créé dans un sac $q \in Q$ ou déplacé à partir d'un sac $q \in Q$. On considère le mot de données dont la projection sur l'alphabet fini est w et dont la valeur de données de chaque position est décrite par le jeton distingué correspondant. L'exécution de l'automate à sac décrit une exécution acceptante de l'automate $\mathcal{A}$ sur le mot de données ainsi obtenu. $\square$

Afin de prouver la décidabilité du vide des automates à sacs de Büchi, on utilise le Lemme de Dickson :

Lemme 2.100 Lemme de Dickson [Dic13] Étant donnée une suite infinie de vecteurs d'entiers $(x_n)_{n\in\mathbb{N}} \subseteq \mathbb{N}^k$ il existe une sous-suite strictement croissante $(i_j)_{j\in Nat}$ telle que $x_{i_1} \leq x_{i_2} \leq \dots$, où la relation $\leq$ est la relation d'ordre composante à composante.

Proposition 2.101 Le vide des automates à sacs de Büchi est décidable.

Preuve. On considère un automate à sacs Büchi $\mathcal{A}$. On commence par montrer que le vide de $\mathcal{A}$ est équivalent à l'existence de deux mots finis u, v, où v est non vide, qui vérifient la propriété $(*)$. On montre alors que ce problème se réduit au vide d'un automate multicompteurs. On obtient ainsi le résultat de décidabilité.

Définition 2.102 Les mots u et v vérifient la propriété $(*)$ s'il existe une exécution ρ sur le mot uv telle que si $i = |u|$ et $j = |uv|$:

1. Les états de l'automate ainsi que l'ensemble des sacs non vides sont identiques aux positions i et j.
2. Étant donné un sac non vide à la position i, chacun de ses jetons passe dans un sac acceptant entre les positions i et j.
3. Tout sac non vide à la position i contient au moins autant de jetons à la position j qu'à la position i.

On commence par montrer que la propriété $(*)$ est une condition nécessaire au non vide de l'automate. Soit w un mot infini accepté par l'automate et ρ une exécution acceptante. Puisque qu'il y a un nombre fini de sacs et d'états, on peut extraire une suite infinie de positions pour lesquelles les états et ensembles de sacs non vides sont les mêmes pour l' exécution ρ. Par le Lemme de Dickson, il existe une sous-suite infinie S strictement croissante de ces positions telle que le nombre de jetons dans chaque sac ne diminue pas. On appelle i la première position de S. Comme l'exécution est acceptante, chaque jeton présent dans un sac à la position i passe par un sac acceptant infiniment souvent. L'ensemble S étant infini, on peut facilement trouver une position de S située après i qui vérifie les propriétés (1-3).

On montre maintenant que la propriété $(*)$ est suffisante pour assurer que l'automate $\mathcal{A}$ est non vide. On considère deux mots u et v qui vérifient la propriété $(*)$ et on montre que le mot uv^ω est accepté par l'automate $\mathcal{B}$. On note $\rho = \rho_u\rho_v$ une exécution de $\mathcal{B}$ sur uv qui vérifie $(*)$ où ρ_u et ρ_v correspondent respectivement aux mots u et v. Les propriétés (1) et (3) de $(*)$ permettent d'affirmer que $\rho_u(\rho_v)^\omega$ est une exécution de $\mathcal{A}$ sur le mot uv^ω. La propriété (1) assure la continuité au niveau des transitions et la propriété (3) assure qu'aucune transition ne cherchera à déplacer un jeton d'un sac vide au cours de l'exécution. La propriété (2) nous donne l'acceptance de l'exécution. Pour chaque sac b non vide après $\rho_u(\rho_v)^k$ on appelle t_b le jeton de b qui n'a pas vu de sac acceptant depuis le plus longtemps. La propriété (2) assure en particulier qu'un jeton de b passe dans un sac acceptant entre $\rho_u(\rho_v)^k$ et $\rho_u(\rho_v)^{k+1}$. Les jetons étant choisis au hasard lors des déplacements, on peut donc en particulier assurer que le jeton t_b passe dans un sac acceptant entre $\rho_u(\rho_v)^k$ et $\rho_u(\rho_v)^{k+1}$. On peut ainsi assurer que chaque jeton passe infiniment souvent dans un sac acceptant.

On a ainsi montré que tout automate à sacs de Büchi non vide accepte un mot ultimement périodique.

On montre maintenant que l'on peut reconnaître, à l'aide d'un automate multicompteurs $\mathcal{M}_{\mathcal{A}}$, les mots de la forme uv où u et v vérifient la propriété $(*)$. Le problème du vide d'un automate à sacs de Büchi est donc réduit au problème du vide pour un automate multicompteurs sur les mots finis.
L'idée est de simuler à l'aide de l'automate $\mathcal{M}_{\mathcal{A}}$ l'exécution de l'automate $\mathcal{A}$ sur les mots uv vérifiant $(*)$. Les compteurs sont utilisés pour vérifier les propriétés sur les sacs. Ainsi on utilise pour chaque sac b de $\mathcal{A}$ quatre compteurs notés b_1, b_2, b_3 et b_4. L'automate $\mathcal{M}_{\mathcal{A}}$ contient donc quatre fois plus de compteurs que $\mathcal{A}$ ne contient de sacs. Une exécution de $\mathcal{A}$ se découpe en deux phases correspondant au découpage du mot d'entrée w en un mot uv vérifiant $(*)$.

Pendant la première phase de l'exécution (correspondant à la partie u de l'entrée) l'automate $\mathcal{M}_{\mathcal{A}}$ simule l'automate $\mathcal{A}$. Le changement d'état suit les transitions de l'automate $\mathcal{A}$. L'évolution des sacs est quant à elle simulée de la manière suivante :

- la création d'un jeton dans le sac b est simulée par l'incémentation des compteurs b_1 et b_2 ;
- le déplacement d'un jeton du sac b au sac b' est simulé par la décrémentation des compteurs b_1 et b_2 suivie de l'incrémentation des compteurs b'_1 et b'_2.

On remarque que pendant cette première phase pour chaque sac b la valeur des compteurs b_3 et b_4 ; les compteurs b_1 et b_2 sont quant à eux égaux et leur valeur correspond à tout moment au nombre de jetons présents dans le sac b dans l'exécution de $\mathcal{A}$.

Le passage à la seconde se fait de manière non-deterministique "l'automate multicompteurs devine qu'il a fini de lire le début du mot correspondant à u" ; on appelle i la position correspondante. Pendant la deuxième phase de l'exécution l'automate $\mathcal{M}_{\mathcal{A}}$ continue à simuler l'exécution de $\mathcal{A}$ sur ce qui correspond au mot v. L'évolution des sacs est maintenant simulée de la manière suivante :

- la création d'un jeton dans le sac b est simulée par l'incémentation des compteurs b_4 ;
- le déplacement d'un nouveau jeton (créé après i) du sac b au sac b' est simulé par la décrémentation du compteur b_4 suivie de l'incrémentation du compteur b'_4 ;
- le déplacement d'un ancien jeton (créé avant i) du sac b au sac b' est simulé par une de deux actions suivantes :
  soit la décrémentation du compteur b_2 suivie de l'incrémentation du compteur b'_3 (resp. b'_2) si b' est (resp. n'est pas) acceptant ;
  soit la décrémentation du compteur b_3 suivie de l'incrémentation du compteur b'_3.

On remarque que pendant cette seconde phase la valeur des compteurs b_1 ne varie pas. La valeur des compteurs se comporte de la manière suivante par rapport à l'exécution correspondante de $\mathcal{A}$:

b_1 = la taille du sac b à la position i (constante).

b_2 = le nombre de jetons actuellement présents dans le sac b qui ont été créés avant la position i et ne sont pas passés dans un sac acceptant depuis i.

b_3 = le nombre de jetons actuellement présents dans le sac b qui ont été créés avant la position i et sont passés dans un sac acceptant depuis i.

b_4 = le nombre de jetons actuellement présents dans le sac b qui ont été créés après la position i.

La condition d'acceptance de l'automate multicompteurs assure que les mots u et v ainsi découpés vérifient à la propriété $(*)$. L'exécution de $\mathcal{M}_{\mathcal{A}}$ sur le mot

uv décrite ci-dessus est acceptante si et seulement si (1) les états simulés de l'automate $\mathcal{A}$ sont les mêmes entre la position i et la fin du mot (correspondant à la position j de la propriété $(*)$) et pour chaque sac b, on a (2) $b_2 = 0$ et (3) $b_1 \leq b_3 + b_4$. La condition (1) et la condition $b_2 = 0$ correspondent à des conditions d'acceptance des automates multicompteurs. Il reste à assurer l'inégalité $b_1 \leq b_3 + b_4$. Cela se fait à l'aide d'une suite d'ϵ-transitions qui décrémentent b_1 et b_3 ou b_1 et b_4 simultanément et de manière non déterministe. À la fin de cette série de décréments, on teste le vide du compteur b_1. □

2.6.2 Extensions indécidables

On présente ici des extensions naturellement intéressantes de la logique qui entraînent malheureusement l'indécidabilité du problème de satisfaisabilité.

FO^3 sans le prédicat d'ordre

Le résultat du Théorème 2.29 dit que la logique du premier ordre est indécidable si on autorise au moins trois variables. La relation $+1$ peut s'exprimer à partir du prédicat $<$ à l'aide de trois variables. On s'intéresse ici au fragment à trois variables n'utilisant pas la relation $<$. On montre que fragment est lui aussi indécidable[22].

Proposition 2.103 Le problème de satisfaisabilité de la logique $\mathrm{FO}^3(\sim, +1)$ est indécidable.

Preuve. On réduit PCP à la satisfaisabilité de $\mathrm{FO}^3(\sim, +1)$. On utilise un codage très proche de celui présenté dans la preuve du Théorème 2.29. Il faut cependant être plus astucieux dans le codage pour assurer que la séquence de données est la même sur les parties Γ et $\bar{\Gamma}$ car on ne dispose plus du prédicat d'ordre $<$ mais simplement du successeur.

Pour des raisons de clarté, on réduit la variante du problème présentée dans le Théorème 2.28 mais la réduction du problème général peut se faire exactement de la même manière. Pour mémoire, la variante consiste à se restreindre aux instances dont les mots d'entrée sont de longueur un ou deux.

On considère une instance I du problème PCP dont les mots d'entrée sont de longueur au plus deux. Le codage des solutions est quasiment identique à celui de la preuve du Théorème 2.29 :

Une solution est codée par un mot de données dont la projection sur l'alphabet fini appartient à $\{u_i\bar{v}_i\}^*$ tel que (1) toutes les positions étiquetées par Γ sont dans des classes différentes, (2) toute classe qui contient une position étiquetée par une lettre $a \in \Gamma$ contient une position étiquetée par $\bar{a}$ et (3) la séquence des valeurs de données sur la partie Γ est la même que celle correspondant à la partie $\bar{\Gamma}$. La Figure 2.3 donne un exemple du codage.

[22] D'après la remarque précédente, ce résultat est plus fort que le Théorème 2.29 qui en devient un corollaire immédiat.

$$I \;=\; \{(a,ab),(br,ra),(ac,c),(a,ad),(da,ab)\}$$

(a) L'instance de PCP

$$u_1\ u_2\ u_3\ u_4\ u_5\ u_2\ u_1 = v_1\ v_2\ v_3\ v_4\ v_5\ v_2\ v_1 = a\ b\ r\ a\ c\ a\ d\ a\ b\ r\ a\ c$$

(b) Une solution

$$\begin{array}{cccccccccccccccccccccccc}
a & \bar{a} & \bar{b} & b & r & \bar{r} & \bar{a} & a & c & \bar{c} & a & \bar{a} & \bar{d} & d & a & \bar{a} & \bar{b} & b & r & \bar{r} & \bar{a} & a & c & \bar{c} \\
d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 & d_4 & d_5 & d_5 & d_6 & d_6 & d_7 & d_7 & d_8 & d_8 & d_9 & d_9 & d_{10} & d_{10} & d_{11} & d_{11} & d_{12} & d_{12}
\end{array}$$

(c) Un codage de cette solution

FIG. 2.3 – Exemple du codage de PCP

Les formules.
On donne maintenant les formules de $\mathrm{FO}^3(\sim,+1)$ qui vont permettre de décrire les solutions d'une instance. On note $\Gamma(x)$ la formule $\bigvee_{a\in\Gamma} a(x)$ et de la même manière on utilise $\bar{\Gamma}(x)$ pour décrire la formule $\bigvee_{a\in\bar{\Gamma}} \bar{a}(x)$.
La projection. Décrire la projection sur l'alphabet fini des solutions se fait à l'aide de la formule $\phi_{u\bar{v}}$.

$$\phi_{u\bar{v}} ::= \forall x\ \#(x) \rightarrow \bigvee_{i\in\{1,\ldots,k\}} \psi_{u_i\bar{v}_i}(x) \wedge \big(fin_{|u_i\bar{v}_i|-1}(x) \vee \#_{|u_i\bar{v}_i|-1}(x)\big)$$

Une formule $\psi_{toto}(x)$ correspond à la formule décrite dans l'Exemple 2.25 qui teste si la projection sur l'alphabet fini du mot prise à partir de la position x commence par le mot *toto*.
La formule $\#(x)$ correspond à la position $\#$ du codage utilisé dans la preuve du Théorème 2.29 qui marque le début des facteurs $u_i\overline{v_i}$ du codage.

$$\#(x) ::= debut(x) \vee \big(\Gamma(x) \wedge \exists y\ (x = y+1 \wedge \overline{\Gamma}(y))\big)$$

Etant donné un entier j on note $\#_j(x)$ la formule qui assure que la position $x+j$ vérifie la formule $\#$. Cette formule se construit par induction comme la formule fin_j décrite dans l'Exemple 2.21 qui assure que $x+j$ est dernière position du mot.
Les classes. La conjonction des deux formules suivantes (notée $\phi_{a\bar{a}}$) assure qu'une classe qui contient une position étiquetée par une lettre de $\Gamma\cup\bar{\Gamma}$ contient exactement deux positions de ce type étiquetées respectivement par une lettre $b\in\Gamma$ et son symétrique $\bar{b}\in\bar{\Gamma}$.

$$\begin{array}{rlcl}
\forall x\forall y & [\Gamma(x)\wedge\Gamma(y)\wedge x\sim y \rightarrow x=y] & \wedge & \big[\bar{\Gamma}(x)\wedge\bar{\Gamma}(y)\wedge x\sim y \rightarrow x=y\big] \\
\forall x & \bigwedge_{b\in\Gamma}\big[b(x)\rightarrow \exists y\ [\bar{b}(y)\wedge x\sim y]\big] & \wedge & \bigwedge_{\bar{b}\in\bar{\Gamma}}\big[\bar{b}(x)\rightarrow \exists y\ [b(y)\wedge x\sim y]\big]
\end{array}$$

Les séquences de données. Pour décrire les séquences de valeurs de données, on utilise le fait que deux positions successives du mot étiquetées par Γ sont à distance bornée. En effet, la projection sur l'alphabet fini du mot de données appartient à $\{u_i\bar{v}_i, i\}^*$ où les u_i et v_i sont de longueur un ou deux. Il en est de même pour les positions étiquetées par $\bar{\Gamma}$. On définit donc deux formules $Succ(x,y)$ et $\overline{Succ}(x,y)$ qui assurent respectivement que x et y sont deux positions successives dans la partie du mot étiquetée par Γ respectivement $\overline{\Gamma}$,avec y

situé à droite de x.

$$\begin{aligned}Succ(x,y) ::= \ & \Gamma(x) \wedge \Gamma(y) \wedge \big[y = x+1 \vee \exists z \ \big[z = x+1 \wedge \overline{\Gamma}(z) \wedge [y = z+1 \\ & \vee \exists x \ [x = z+1 \wedge \bar{\Gamma}(x) \wedge y = x+1]]]\big]\big]\end{aligned}$$

$$\begin{aligned}\overline{Succ}(x,y) ::= \ & \overline{\Gamma}(x) \wedge \overline{\Gamma}(y) \wedge \big[y = x+1 \vee \exists z \ \big[z = x+1 \wedge \Gamma(z) \wedge [y = z+1 \\ & \vee \exists x \ [x = z+1 \wedge \Gamma(x) \wedge y = x+1]]]\big]\big]\end{aligned}$$

Il ne reste plus qu'à assurer l'égalité des séquences de données entre la partie Γ et la partie $\bar{\Gamma}$. On utilise pour cela la formule ϕ_{seq} qui impose que les classes de deux positions successives de la partie Γ apparaissent dans le même ordre dans deux positions successives de la partie $\bar{\Gamma}$.

$$\phi_{seq} ::= \forall x \forall y \ Succ(y,x) \ \rightarrow (\exists z \ \ x \sim z \wedge (\exists x \ \overline{Succ}(z,x) \wedge x \sim y))$$

Bilan : La formule ϕ_I est la conjonction d'une formule $\phi_{u\bar{v}}$ qui assure la structure de la projection sur l'alphabet fini, une formule $\phi_{a\bar{a}}$ qui assure que toute classe qui contient une lettre contient exactement deux positions étiquetée par $\Gamma \cup \bar{\Gamma}$ qui correspondent à une paire $(a, \bar{a})$ et une formule ϕ_{seq} qui assure que la suite des données est la même sur les parties Γ et $\bar{\Gamma}$. On peut remarquer que les seules formules qui utilisent trois variables sont les formules $Succ$ et $\overline{Succ}$.

Par construction les modèles de cette formule sont exactement les codages des solutions de l'instance I de PCP. Le problème de satisfaisabilité des formules de $\mathrm{FO}^3(\sim, +1)$ est donc indécidable. $\square$

Deux relations d'équivalence

Une autre idée est d'étendre les modèles considérés de sorte que chaque position du mot stocke un vecteur de valeurs de données. L'idée vient entre autre des documents XML où chaque nœud peut avoir plusieurs attributs. Si on voit les données comme une relation d'équivalence sur le mot, cela revient à munir la logique de plusieurs prédicats d'égalité de données $\sim_1$, $\sim_2, \ldots$ chacun d'entre eux permettant de tester l'égalité des données pour une des composantes. Malheureusement, les résultats présentés dans [KO05] montrent que la logique FO^2 est indécidable si on autorise trois relations d'équivalence et des prédicats unaires. On montre ici que la satisfaisabilité de la logique est indécidable si on autorise deux relations d'équivalence, l'ordre linéaire et le successeur ou deux relations d'équivalence et les prédicats successeurs $+1, +2, +3$.

Il est néanmoins possible d'obtenir un résultat de décidabilité en considérant des modèles avec plusieurs relations d'équivalences. Dans [BB07b], les auteurs montrent un résultat de décidabilité de la logique FO^2 avec plusieurs relations d'équivalence reliées par une relation d'ordre. Cette relation impose que toute relation d'équivalence $\sim$ inférieure à $\sim'$ est un raffinement de la première.

Proposition 2.104 Le problème de satisfaisabilité est indécidable pour la logique $\mathrm{FO}^2(\sim_1, \sim_2, <, +1)$.

Preuve. Dans la suite on parle de *mot de 2-données* pour désigner les mots de $(\Sigma \times \mathcal{D} \times \mathcal{D})^*$, de $\sim_i$ pour désigner la relation d'équivalence correspondant à la i-ième valeur de données des positions du mot, et de i-classe pour désigner une classe de $\sim_i$ (avec $i = 1$ ou 2).

On prouve le résultat par une réduction du Problème de Correspondance de Post (PCP) restreint aux mots de longueur un ou deux.

Étant donnée une instance de PCP, on donne un codage des solutions du problème. On construit alors une formule $\varphi_I \in \mathrm{FO}^2(\sim_1, \sim_2, <, +1)$ qui reconnaît exactement les codages des solutions de l'instance I.

On considère $I = (u_i, v_i)_{i \in [1..k]}$ une instance de PCP et $j_1, \ldots, j_n$ une solution.

On choisit un alphabet $\Sigma = \Gamma \cup \bar{\Gamma}$ où $\bar{\Gamma}$ est une copie de l'alphabet[23] Γ. On code une solution dans un mot de données qui vérifie les propriétés suivantes :

(1) La projection sur l'alphabet fini du mot est $u_{j_1}\overline{v_{j_1}}u_{j_2}\overline{v_{j_2}} \ldots u_{j_n}\overline{v_{j_n}}$ où $\bar{v}_i$ représente le mot v_i écrit avec l'alphabet $\bar{\Gamma}$.

(3) La séquence de *paires de données* sur la partie Γ est exactement la même que sur la partie $\bar{\Gamma}$.

(4) Cette séquence est de la forme $(\alpha_1, \beta_1)(\alpha_1, \beta_2)(\alpha_2, \beta_2)(\alpha_2, \beta_3) \ldots (\alpha_{m-1}, \beta_m)(\alpha_m, \beta_m)$ où tous les α_i (resp. β_j) sont différents.

Intuitivement chaque position du mot $u_{i_{j_1}} u_{i_{j_2}} \ldots u_{i_{j_n}} = v_{i_{j_1}} v_{i_{j_2}} \ldots v_{i_{j_n}}$ correspond à une paire de valeurs de données différentes.

$$I \;=\; \{(a, ab), (br, ra), (ac, c), (a, ad), (da, ab)\}$$

(a) L'instance de PCP

$$u_1\ u_2\ u_3\ u_4\ u_5\ u_2\ u_3 = v_1\ v_2\ v_3\ v_4\ v_5\ v_2\ v_3 = a\ b\ r\ a\ c\ a\ d\ a\ b\ r\ a\ c$$

(b) Une solution

a	$\overline{a}$	$\overline{b}$	b	r	$\overline{r}$	$\overline{a}$	a	c	$\overline{c}$	a	$\overline{a}$	$\overline{d}$	d	a	$\overline{a}$	$\overline{b}$	b	r	$\overline{r}$	$\overline{a}$	a	c	$\overline{c}$
α_1	α_1	α_1	α_1	α_2	α_2	α_2	α_2	α_3	α_3	α_3	α_3	α_4	α_4	α_4	α_4	α_5	α_5	α_5	α_5	α_6	α_6	α_6	α_6
β_1	β_1	β_2	β_2	β_2	β_2	β_3	β_3	β_3	β_3	β_4	β_4	β_4	β_4	β_5	β_5	β_5	β_5	β_6	β_6	β_6	β_6	β_7	β_7

(c) Un codage de cette solution

FIG. 2.4 – Exemple du codage de PCP

Un mot de 2-données est le codage d'une solution si et seulement s'il vérifie les propriétés suivantes :

(a) La projection sur l'alphabet fini du mot appartient au langage ${\{u_i\bar{v}_i\}^*}_{i \in [1..k]}$.

(b) Toute 1-classe contient exactement deux positions étiquetées dans Γ et deux positions étiquetées par $\bar{\Gamma}$ sauf la classe correspondant à la première position du mot qui contient exactement une position étiquetée Γ et une position étiquetée $\bar{\Gamma}$.

(c) Symétriquement, toute 2-classe contient exactement deux éléments étiquetés dans Γ et deux positions étiquetées par $\bar{\Gamma}$ sauf la classe correspondant à la dernière position du mot qui contient exactement une position étiquetée Γ et une position étiquetée $\bar{\Gamma}$.

(d) Pour toute paire de données qui correspond à une position étiquetée par un $a \in \Gamma$, il existe exactement une position qui contient la même paire de données étiquetée $\bar{a}$. Réciproquement, pour toute paire de donnée qui correspond à une position étiquetée par un $\bar{a} \in \bar{\Gamma}$, il existe exactement une position qui contient la même paire de données étiquetée a.

[23] Si a est une lettre de Γ, on note $\bar{a}$ sa copie dans $\bar{\Gamma}$

(e) Pour toute position étiquetée par Γ sauf la première et la dernière, on est dans une des deux situations suivantes :
  - il existe une Γ-position à sa droite dans la même 1-classe et une Γ-position à sa gauche dans la même 2-classe,
  - il existe une Γ-position à sa droite dans la même 1-classe et une Γ-position à sa gauche dans la même 2-classe,

  On a la même propriété pour les positions étiquetées par $\bar{\Gamma}$.

Par construction, un mot de 2-données qui est une solution de I vérifie (a-e). Réciproquement si un mot de 2-données vérifie les propriétés (a-e), il vérifie les propriétés (1-5) et donc est un modèle d'une solution du problème. La propriété (1) est une conséquence de (a). La propriété (d) assure que les parties Γ et $\bar{\Gamma}$ du mot sont de la même longueur. On montre les propriétés (2) et (3) par induction sur cette longueur (notée l).
Le cas $l = 1$ est immédiat par la propriété (d). Le cas $k+1$ se montre de la manière suivante : d'après la propriété (e), $x_{k-1} \sim_1 x_k$ ou $x_{k-1} \sim_2 x_k$; on suppose que $x_{k-1} \sim_1 x_k$, l'autre cas étant symétrique. Par hypothèse d'induction, les suites de données étant identiques sur les parties Γ et $\bar{\Gamma}$, on a aussi $\bar{x}_{k-1} \sim_1 \bar{x}_k$ et les x_k et $\bar{x}_k$ sont dans les mêmes 1-classes et 2-classes. La propriété (e) impose que $x_k \sim_2 x_{k+1}$ et $\bar{x}_k \sim_2 \bar{x}_{k+1}$. Les positions x_{k+1} et $\bar{x}_{k+1}$ sont donc dans la même 2-classe. Il reste à montrer qu'elles sont dans la même 1-classe et que celle-ci correspond à une nouvelle donnée. La propriété (d) combinée aux propriétés (b) et (c) assure que les deux positions sont dans la même 1-classe. Par l'hypothèse d'induction et les propriétés (b) et (c), cette 1-classe correspond bien à une nouvelle classe.

Il reste à montrer que les propriétés s'expriment à l'aide de formules de la logique $\mathrm{FO}^2(\sim_1, \sim_2, <, +1)$. Les propriétés (a), (b), (c) et (d) utilisent des formules[24] similaires à celles utilisées dans la preuve de la Proposition 2.103. La propriété (e) pour la partie Γ s'exprime facilement de la manière suivante :

$$\begin{array}{lcl} \forall x\, \Gamma(x) & \rightarrow & (\exists y\, \Gamma(y) \wedge\ y \sim_1 x \ \wedge\ y < x) \ \wedge\ (\exists y\, \Gamma(y)\ \wedge y \sim_2 x \ \wedge\ x < y) \\ & \vee & (\exists y\, \Gamma(y) \wedge\ y \sim_1 x \ \wedge\ x < y) \ \wedge\ (\exists y\, \Gamma(y)\ \wedge y \sim_2 x \ \wedge\ y < x) \end{array}$$

□

Proposition 2.105 Le problème de satisfaisabilité pour $\mathrm{FO}^2(\sim_1, \sim_2, +1, +2, +3)$ est indécidable.

Preuve. Si on interdit le prédicat $<$ dans les formules, on peut montrer le même résultat en utilisant le codage précédent et, comme dans la preuve de la Proposition 2.103, le fait que deux positions consécutives de Γ ou $\bar{\Gamma}$ sont à distance au plus trois. A l'aide des prédicats successeurs $+1$, $+2$ et $+3$ on peut définir les formules *Succ* et $\overline{Succ}$ en utilisant seulement deux variables. On peut ainsi contraindre l'égalité des séquences de paires de données.

$$\begin{array}{rl} Succ(x,y) ::= & \Gamma(x) \wedge \Gamma(y) \wedge \big[y = x+1 \\ & \vee[y = x+2 \wedge \exists y\ [y = x+1 \wedge \overline{\Gamma}(y)]] \\ & \vee[y = x+3 \wedge \exists y\ [y = x+1 \wedge \overline{\Gamma}(y)] \wedge \exists y\ [y = x+2 \wedge \overline{\Gamma}(y)]]\big] \end{array}$$

[24] Dans la preuve de la Proposition 2.103, on assure ce type de propriétés à l'aide de formules de $\mathrm{FO}^2(\sim, +1)$

$$\begin{aligned}\overline{Succ}(x,y) ::= \;& \bar{\Gamma}(x) \wedge \bar{\Gamma}(y) \wedge [y = x+1 \\ & \vee[y = x+2 \wedge \exists y\ [y = x+1 \wedge \Gamma(y)]] \\ & \vee[y = x+3 \wedge \exists y\ [y = x+1 \wedge \Gamma(y)] \wedge \exists y\ [y = x+2 \wedge \Gamma(y)]]]\end{aligned}$$

On modifie légèrement la propriété (e) car on ne dispose pas de la relation d'ordre mais on va utiliser la relation $Succ$ et $\overline{Succ}$

Un mot de 2-données est le codage d'une solution si et seulement s'il vérifie les propriétés suivantes :

(a) La projection sur l'alphabet fini du mot appartient au langage $\{u_i\bar{v}_i\}^*{}_{i\in[1..k]}$.

(b) Toute 1-classe contient exactement deux positions étiquetées dans Γ et deux positions étiquetées pas $\bar{\Gamma}$ sauf la classe correspondant à la première position du mot qui contient exactement une position étiquetée Γ et une position étiquetée $\bar{\Gamma}$.

(c) Symétriquement, toute 2-classe contient exactement deux éléments étiquetés dans Γ et deux positions étiquetées pas $\bar{\Gamma}$ sauf la classe correspondant à la dernière position du mot qui contient exactement une position étiquetée Γ et une position étiquetée $\bar{\Gamma}$.

(d) Pour toute paire de données qui correspond à une position étiquetée par un $a \in \Gamma$, il existe exactement une position qui contient la même paire de données étiquetée $\bar{a}$. Réciproquement, pour toute paire de donnée qui correspond à une position étiquetée par un $\bar{a} \in \bar{\Gamma}$, il existe exactement une position qui contient la même paire de données étiquetée a.

(e) Pour toute position étiquetée par Γ sauf la première et la dernière, on est dans une des deux situations suivantes :
- Son successeur dans la partie Γ est dans la même 1-classe et son prédécesseur dans la partie Γest dans la même 2-classe.
- Son prédécesseur dans la partie Γ est dans la même 1-classe et son successeur dans la partie Γ est dans la même 2-classe.

On a la même propriété pour les positions étiquetées par $\bar{\Gamma}$.

La seule propriété qui demande une explication supplémentaire est la propriété (e). On présente la formule ϕ_Γ qui correspond aux positions Γ. La formule $\phi_{\mathrm{Fst}\Gamma}$ permet de tester si une position est la première position Γ du mot. Les formules ϕ_1 et ϕ_2 décrivent les deux cas possibles. Ces formules s'appuient évidemment sur le fait que les propriétés (a-d) soient vraies.

$$\begin{aligned}\phi_{\mathrm{Fst}\Gamma}(x) ::= \;& \Gamma(x) \wedge \forall y\ [\neg\Gamma(y) \vee x < y] \\ \phi_1(x) ::= \;& \exists y\ [x \sim_{1,2} y \wedge \exists x\ [Succ(y,x) \wedge y \sim_1 x \wedge y \nsim_2 x]] \\ \wedge \;& \exists y\ [x \sim_{1,2} y \wedge \exists x\ [Succ(x,y) \wedge y \nsim_1 x \wedge y \sim_2 x]] \\ \\ \phi_2(x) ::= \;& \exists y\ [x \sim_{1,2} y \wedge \exists x\ [Succ(y,x) \wedge y \nsim_1 x \wedge y \sim_2 x]] \\ \wedge \;& \exists y\ [x \sim_{1,2} y \wedge \exists x\ [Succ(x,y) \wedge y \sim_1 x \wedge y \nsim_2 x]] \\ \\ \phi_\Gamma ::= \;& \forall x\ [\Gamma(x) \wedge \neg\phi_{\mathrm{Fst}\Gamma}(x)] \rightarrow [\phi_1(x) \vee \phi_2(x)]\end{aligned}$$

On construit de la même manière une formule $\phi_{\bar{\Gamma}}$. □

Ordre linéaire sur les données

Une autre façon d'étendre les modèles et la logique est de munir l'ensemble $\mathcal{D}$ des valeurs de données d'une relation d'ordre $\preccurlyeq$ et d'augmenter la logique par un prédicat binaire correspondant (que l'on note aussi $\preccurlyeq$). On montre que cette simple opération entraîne l'indécidabilité de la logique.

Proposition 2.106 Le problème de satisfaisabilité de $\mathrm{FO}^2(\sim, \preccurlyeq, +1, <)$ est indécidable.

Preuve. On reprend la preuve du Théorème 2.29. On utilise le même codage de PCP en imposant que la séquence de données apparaissant dans les parties Γ et $\bar{\Gamma}$ est une suite strictement croissante de données. La seule formule de cette preuve qui utilise trois variables est la formule qui assure que toute paire de données apparaît dans le même ordre dans la partie Γ et dans la partie $\bar{\Gamma}$. On remplace cette formule par une formule qui assure que dans chaque partie du mot la suite des données est strictement croissante.

$$\forall x, y\ \Gamma(x) \wedge \Gamma(y) \wedge x < y \rightarrow x \preccurlyeq y$$

(on a la même formule pour $\overline{\Gamma}$). Cette formule à elle seule assure d'une part la nouvelle contrainte de monotonie et l'égalité des séquence des données (en association avec les autres formules bien-sûr). $\square$

2.7 Conclusion

On a présenté une extension de la logique du premier ordre aux mots de données en utilisant un prédicat binaire $\sim$ pour comparer les valeurs de données des positions d'une formule. Le problème de satisfaisabilité est indécidable en général (Théorème 2.29) mais on montre que le fragment restreint aux formules utilisant deux variables $\mathrm{FO}^2(\sim, < +1)$ est décidable.

Dans le cas des mots de données une extension des automates po-2-DFA avec un registre est proposée dans [Dav04]. Malheureusement, le problème du vide est indécidable pour cette classe d'automate.

Afin de montrer la décidabilité de $\mathrm{FO}^2(\sim, <, +1)$, on définit une nouvelle classe d'automates sur les mots de données appelés **automates à données**. On montre dans le Théorème 2.90 l'équivalence de ces automates avec le fragment étendu $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$. On obtient une vision alternative des résultats présentés dans la Section 2.1 qui propose une extension de la logique temporelle et des automates sur les mots de données à l'aide de registres.

Sans valeurs de données, la logique $\mathrm{FO}^2(<, +1)$ est équivalente à la logique $\mathrm{LTL}(\mathtt{X}, \mathtt{X}^{-1}, \mathtt{F}, \mathtt{F}^{-1})$. Dans le cas des logiques sur les mots de données, on perd cette équivalence. Les fragments décidables $\mathrm{FO}^2(\sim, <, +1)$ et $\mathrm{LTL}_1^{\downarrow}(\mathtt{X}, \mathtt{U})$ sont incomparables. En effet, la logique temporelle n'autorise que des modalités futur donc elle ne peut pas exprimer de propriété d'inclusion comme par exemple "Toute classe qui contient un a contient aussi un b" alors que ce type de propriété s'exprime naturellement avec la logique FO^2. Réciproquement, l'opérateur $\mathtt{U}$ de la logique temporelle permet de décrire des propriétés que la restriction à deux variables des formules FO^2 ne permet pas d'exprimer. Par exemple, comme on le verra dans le Lemme 3.31, on ne peut pas assurer l'existence de deux positions de la même classe séparée par un position étiquetée b. Ou encore, il ne semble pas possible d'exprimer le fait qu'entre des positions a de la même classe il existe toujours exactement une position b.

Il existe cependant des liens entre ces deux logiques [DL08a]. Une formule de $\mathrm{LTL}_1^{\downarrow}(O)$ est dite “simple” si et seulement si toute occurrence d'un opérateur temporel est précédé de l'opérateur $\downarrow_1$ (qui n'apparaît nulle part ailleurs). Le Théorème 3.2 de [DL08a] montre qu'il y a une équivalence entre les formules simples de $\mathrm{LTL}_1^{\downarrow}(\mathtt{X}, \mathtt{X}^{-1}, \ldots, \mathtt{X}^m, \mathtt{X}^{-m}, \mathtt{X}^{m+1}\mathtt{F}, \mathtt{X}^{-(m+1)}\mathtt{F})$ et les formules de $\mathrm{FO}^2(\sim, <, +1, \ldots, +m)$. Les transformations sont effectives LogSpace de LTL vers FO^2 et PSpace de FO^2 vers LTL. La satisfaisabilité des fragments $\mathrm{FO}^2(\sim, <, +1, \ldots, +m)$ est décidable sur les mots finis (Théorème 2.92) et sur les mots infinis (en étendant la preuve du Théorème 2.94 aux successeurs étendus). On obtient ainsi pour la logique $\mathrm{LTL}_1^{\downarrow}$, des fragments qui autorisent des opérateurs passés tout en restant décidables.

Un autre aspect très intéressant de ce travail est le fort lien entre la logique $\mathrm{FO}^2(\sim, <, +1)$ et les automates multicompteurs (modèle équivalent aux réseaux de Petri). La Proposition 2.58 montre l'équivalence entre ces deux modèles. D'une part, le problème de satisfaisabilité de $\mathrm{FO}^2(\sim, <, +1)$ se réduit en temps doublement exponentiel au problème du vide des automates multicompteurs . D'autre part le problème du vide pour ces automates se réduit en temps polynomial au problème de satisafaisabilité. Il serait intéressant d'approfondir cette étude afin de mieux comprendre la difficulté en complexité de ces deux modèles. En particulier, la réduction de la décidabilité des formules vers le problème du vide des automates multicompteurs est 2ExpTime. On ne sait pas si cette réduction est optimale.

En matière de complexité, les remarques précédente n'est pas encourageante. Le problème de satisfaisabilité de la logique $\mathrm{FO}^2(\sim, +1, <)$ ainsi du même ordre de difficulté que le problème d'accessibilité dans les réseaux de Petri. On ne connaît pas de borne de complexité élémentaire et la meilleure borne inférieure connue est ExpSpace-dur.

Si on restreint l'expressibilité logique, on obtient des bornes de complexité élémentaires. Le Théorème 2.59 montre que le problème de satisfaisabililté est NExpTime-complet pour la logique $\mathrm{FO}^2(\sim, <)$. Dans le cas de la logique $\mathrm{FO}^2(\sim, +1)$, le Théorème 2.63 montre que le problème est 3NExpTime. La meilleure borne inférieure connue pour ce problème est NExpTime, il serait intéressant de connaître la complexité exacte du problème.

De nombreuses extensions de la logiques $\mathrm{FO}^2(\sim, <)$ proposées mènent à l'indécidabilité : trois variables (Théorème 2.29 et Proposition 2.103), deux relations d'équivalence (Proposition 2.104 et 2.105), relation ordre sur les données (Proposition 2.106).

Il est aussi intéressant de remarquer que sans valeurs de données, la logique $\mathrm{EMSO}^2(+1, <)$, la logique monadique du second ordre (MSO) et les automates finis ont la même expressivité. Dans le cas des mots de données, la logique monadique du second ordre munie du prédicats d'égalité de données est indécidable. Cependant, on retrouve l'équivalence entre la logique $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$ qui est décidable et équivalente, en terme d'expressivité, à la classe des automates à données.

Une perspective de tous ces travaux est de considérer l'extension des logiques et automates aux arbres de données. Ceci fait l'objet du chapitre suivant.

Chapitre 3

Arbres de données : applications/relations avec XML.

Dans ce chapitre, on s'intéresse plus particulièrement aux cas des arbres de données. La première section est une présentation du langage XML et des problématiques associées. Les deux sections suivantes expliquent dans quelle mesure on peut étendre les formalismes du chapitre précédent au modèle des arbres de données. Dans la dernière section, on revient au langage XML afin de relier les résultats sur les arbres des Sections 3.2 et 3.3 aux problématiques présentées au début de ce chapitre.

3.1 XML et données

Les documents XML

Le langage XML (eXtended Markup Language) [W3C06] apparaît comme une réponse au besoin de structures de données plus flexibles que les bases de données relationnelles dans le contexte distribué, hétérogène et évolutif de l'Internet. Le but de ce langage est à l'origine de faciliter l'échange et le partage d'information. Le langage XML vient d'une simplification du langage de description SGML (Standard Generalized Markup Language) [W3C]. Il permet de représenter des données de manière structurée à l'aide de *balises* sous la forme d'un arbre.

Depuis 1998, XML est un format standard d'échange de données et est largement utilisé par exemple dans les applications de commerce en ligne, pour la diffusion d'information via les flux rss, ou encore dans les bibliothèques en ligne. La flexibilité de sa structure en fait un langage utile à la communauté des bases de données pour décrire des données semi-structurées hors du contexte de l'Internet. On utilise ainsi des documents XML pour la représentation interne de structure de données dans OpenOffice.org, dans les fichiers de configuration de certaines applications, ou encore pour représenter des listes de lecture dans des logiciels de gestion de musique comme Rhythmbox ou iTunes.

La figure suivante donne un exemple de document XML qui représente le contenu d'une bibliothèque.

```
<bibliothèque>
    <livre isbn="210003202" >
        <titre lng=fr>
            "Le livre qui rend fou"
        </titre>
        <auteurs>
            <personne>
                <prénom>
                    "Raymond"
                </prénom>
                <nom>
                    "Smullyan"
                </nom>
            </personne>
        </auteurs>
        <résumé lng=fr>
            "Livre d'énigmes logiques"
        </résumé>
    </livre>

    <livre isbn="0870408526">
        <titre>
            "Unit Origami"
        </titre>
        <auteurs>
            <personne>
                <prénom>
                    "Tomoko"
                </prénom>
                <nom>
                    "Fusè"
                </nom>
            </personne>
        </auteurs>
        <résumé lng=fr>
            "Livre d'origami modulaire"
        </résumé>
        <résumé lng=jap>
            ...
        </résumé>
    </livre>
</bibliothèque>
```

FIG. 3.1 – Document XML représentant le contenu d'une bibliothèque

Un document XML est constitué d'éléments, d'attributs et de données textuelles. Les balises <...>, et </..> délimitent respectivement le début et la fin d'un élément. Un élément est associé à un tag, dans l'exemple précédent `livre`, `personne` ou encore `resume` sont des tags. Les attributs permettent de décrire des propriétés associées à un élément ; un attribut associé à un élément contient ainsi une valeur a priori quelconque. Dans l'exemple précédent, `isbn` est un attribut, il contient la valeur `210003202` dans le cas du premier élément `livre`. Un élément peut contenir d'autres éléments ainsi que des données textuelles. On reprend dans l'exemple précédent, l'élément correspondant au premier livre. Il s'agit d'un élément `livre` contenant un attribut `isbn` dont la valeur est "*210003202*". Cet élément contient un élément `titre` qui contient la donnée textuelle "*Le livre qui rend fou*", un élément `personne` et un élément `resume`

Une condition de bonne formation des documents XML est le bon parenthésage des balises ouvrantes et fermantes. On peut donc voir un document XML comme un arbre à travers ces balises. La Figure 3.2 donne la représentation standard d'une partie du document de l'exemple comme un arbre. Un tel arbre est d'arité non bornée et a priori ordonné. Un élément est représenté par un nœud étiqueté par le nom du tag correspondant. De la même manière, un attribut est représenté par un nœud étiqueté par son nom (en pointillés dans la figure). Les données (texte et valeur d'attribut) sont stockées dans les feuilles. Dans certains documents l'ordre horizontal entre les nœuds n'a pas de sens du point de vue sémantique, on peut alors voir le document comme un arbre non ordonné (on reviendra sur cette hypothèse dans la Section 6.4).

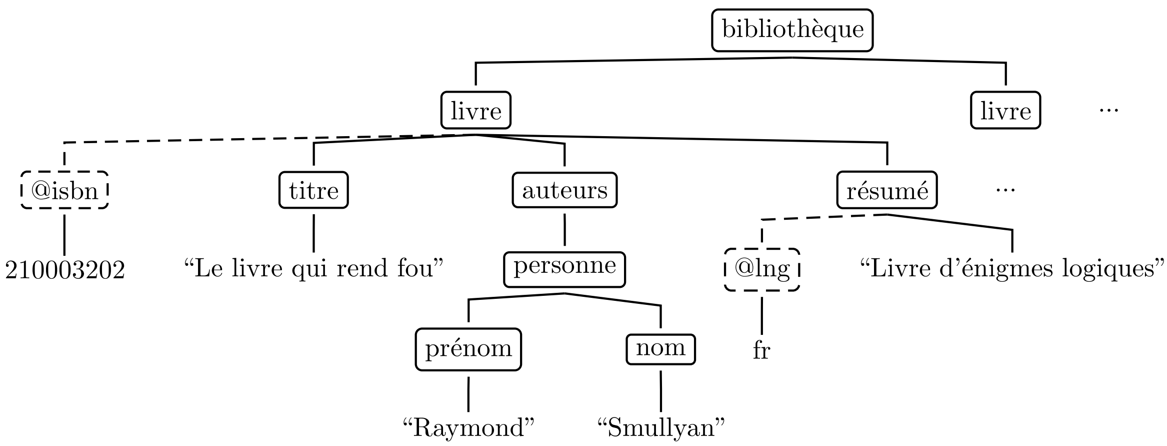


FIG. 3.2 – Un document XML vu comme un arbre

Sémantique

La structure des documents étant flexible, un document XML peut a priori représenter n'importe quel arbre. Cependant il est bien évident que dans un contexte d'utilisation, on s'intéresse à un sous-ensemble de documents particuliers. On a donc besoin de formalismes afin de pouvoir définir ce qu'on appelle des *types de document* ou des *schémas*. Les contraintes imposées aux documents à travers un schéma sont une manière de préciser la sémantique de l'information représentée par les documents pour l'application donnée.

```
<!ELEMENT bibliotheque (livre*)>
<!ELEMENT livre (titre,auteurs,resume?)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteurs (personne*)>
<!ELEMENT personne (prenom,nom)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT resume (#PCDATA)>
<!ATTLIST livre isbn ID #REQUIRED>
<!ATTLIST resume lng CDATA #IMPLIED>
```

FIG. 3.3 – Exemple de DTD

Dans ce cadre, on peut faire une distinction entre les tags qui représentent la structure de l'arbre et les données associées (valeurs d'attribut et données textuelles). Un façon naturelle de définir un type de document est de contraindre la structure des documents. Mais selon le contexte, on veut aussi pouvoir imposer des contraintes sur les données afin de raffiner la description. Par exemple, lorsque l'on décrit un type de document correspondant à une bibliothèque, il est naturel de vouloir imposer le fait que tous les livres ont des numéros ISBN différents. On appelle cette propriété une contrainte de clé.

Les contraintes que l'on impose en général aux documents concernent principalement la structure de l'arbre ainsi que les tags mais considérent très peu les données textuelles ou encore les valeurs d'attributs. Dans les langages de typage les plus utilisés (DTD, XMLSchema) ces contraintes imposent ainsi que la structure finie du document appartienne à un langage régulier d'arbres (étiqueté par un ensemble fini de tags) décrit par exemple par une grammaire d'arbres.

Les DTDs (Document Type Definition) constituent le moyen le plus simple et le plus fréquemment utilisé pour décrire la structure des documents. Une DTD décrit pour chaque tag l'ensemble des successeurs possibles sous la forme d'une expression régulière simple. La figure 3.3 donne un exemple de DTD. Les différents indicateurs d'occurrence utilisés dans les DTDs sont "$*$" qui signifie 0 ou plusieurs éléments, "+" pour au moins un élément et "?" qui signifie 0 ou 1 élément. Le type "#PCDATA" signifie qu'un élément contient des données. La première ligne décrit la racine de l'arbre, dans l'exemple, un élément `bibliotheque` qui peut contenir une suite éventuellement vide d'éléments `livre`. La deuxième ligne décrit les éléments `livre` ; un tel élément contient un élément `titre`, un élément `auteurs` et un éventuel élément `resume`. Même si dans de nombreuses situation une DTDs suffit à décrire la structure générale d'un document. Si on ne considère que la structure finie, les DTDs ont un pouvoir expressif restreint, en particulier on ne peut pas définir deux expressions régulières différentes pour un même type d'élément.

Si on considère uniquement la structure des documents, on peut voir un schéma comme la description d'un langage régulier d'arbres ordonnés d'arité non bornée étiquetés sur un alphabet fini de tags.

Cependant, parler uniquement de la structure des documents n'est pas toujours suffisant pour la description d'un ensemble de documents. On veut aussi rajouter de la sémantique au niveau des données pour "raffiner" les schémas.

Si on reprend l'exemple précédent de la bibliothèque, on veut imposer que l'attribut `isbn` est une clé. Les deux dernières lignes de l'exemple imposent à tout élément `livre` d'avoir un attribut `isbn` (cela correspond à la mention "`#REQUIRED`"). La mention "`ID`" impose que l'attribut soit une clé.

Définition 3.1 On dit qu'un ensemble d'attributs X est une clé [FL02] pour le tag τ dans un document si et seulement si le document ne contient pas deux éléments τ qui ont le même tuple de valeurs pour cet ensemble X d'attributs. On note cette contrainte $X \rightarrow \tau$. Si l'ensemble contient un seul attribut, on parle de clé unaire.

Imaginons que l'on enrichisse l'exemple précédent de bibliothèque en incluant ce document dans un document plus général qui va contenir aussi le liste des abonnés de la bibliothèque. Une manière raisonnable de représenter un abonné est de le représenter par un élément `abonne` contenant un attribut `numero` qui contient son numéro d'abonné et un élément supplémentaire. Cet élément peut ensuite contenir toutes sortes d'informations comme le nom et les coordonnées de l'abonné. Il est naturel d'imposer que l'attribut `numero` soit une clé pour l'élément `abonne`. On peut alors ajouter à chaque livre un élément `pret` lorsque le livre a été emprunté. On associe à ces éléments un attribut `emprunteur` (censé représenter l'abonné qui a emprunté le livre). Il est naturel d'imposer que toute valeur de l'attribut `emprunteur` qui apparaît dans le document apparaisse aussi comme valeur de l'attribut `numero` d'un élément `abonne`. Ce type de contrainte correspond à la notion d'inclusion et de clé étrangère [FL02].

Définition 3.2 Étant donnés deux tags τ_1 et τ_2 et deux listes d'attributs X et Y de même longueur, on dit qu'un document vérifie la contrainte d'inclusion notée $\tau_1[X] \subseteq \tau_2[Y]$ si et seulement si pour tout élément τ_1 du document ayant les attributs X, il existe un élément τ_2 avec les attributs Y tel que les tuples des valeurs d'attribut sont égaux.

Définition 3.3 Une clé étrangère est la conjonction d'une contrainte de clé $Y \rightarrow \tau_2$ et d'une contrainte d'inclusion $\tau_1[X] \subseteq \tau_2[Y]$.

De manière plus générale, on parle de *contrainte d'intégrité* pour désigner un ensemble de contraintes de type clé, clé étrangère ainsi que des négations de telles contraintes. Les DTDs permettent d'imposer des contraintes de type clé et clé étrangère unaire.

Mais parler des attributs n'est pas la seule manière de contraindre les données : on peut imaginer des contraintes sur les données textuelles. Une manière simple de voir cela est de remplacer dans la représentation précédente l'*attribut* `numero` par un *élément* `numero`. La valeur de l'attribut devient alors une donnée textuelle. On veut maintenant exprimer le fait que les données stockées dans les éléments `numero` qui sont contenus dans les éléments `abonne` sont toutes différentes. La contrainte de clé s'exprime maintennant sur la donnée textuelle. De manière plus générale, on veut pouvoir imposer une contrainte de clé sur un ensemble d'éléments plus complexe. On a alors besoin de caractériser des ensembles d'éléments dans un document.

Requêtes

Une raison pour laquelle on veut pouvoir caractériser des ensembles de nœuds est pour manipuler ces documents et extraire de l'information. Un premier exemple est la transformation de document. Imaginons que notre bibliothèque fusionne avec une bibliothèque qui stockait, en plus de la liste des livres, la liste des auteurs en faisant référence à cette liste dans les éléments correspondant aux livres. On voudrait pouvoir transformer automatiquement le document correspondant à notre bibliothèque afin qu'il respecte le schéma de la nouvelle bibliothèque. Il va falloir par exemple extraire l'ensemble des éléments correspondant aux auteurs afin de les ajouter à la liste déjà existante (si possible en évitant les doublons). Une fois les deux bases de données fusionnées (et les livres rangés), on veut permettre aux abonnés de la bibliothèque d'effectuer des recherches dans cette base de données. De manière générale, que ce soit pour utiliser cette information de façon automatique afin de transformer les documents (par exemple à l'aide du langage XSLT [W3C99b]) ou pour répondre à des questions d'utilisateurs (XQuery [W3C07]), il est nécessaire de pouvoir naviguer dans l'arbre et de pouvoir sélectionner des nœuds du document. Le langage XPath [W3C99a] permet de naviguer et sélectionner un ensemble d'éléments dans un document. C'est sur la base de ce langage que sont construits à la fois le langage XSLT utilisé pour la transformation de documents et le langage de requête XQuery.

On ne présente ici qu'un fragment (appelé Data-Core-Xpath) représentant l'essentiel du langage XPath qui nous sera utile par la suite. On réfère le lecteux aux deux articles suivant pour plus de détails [BFG05, GF05]. On appelle $\Sigma = T \uplus Attr$, l'alphabet constitué d'un ensemble fini de tags T et des attributs $Attr$.

Définition 3.4 Une requête XPath est définie par la grammaire suivante :

$$p ::= \ Axis :: a \mid p/p \mid p \cup p \mid p \cap p \mid p[q]$$

$$q ::= \ \neg\, q \mid q \wedge q \mid q \vee q \mid p \mid p/@A \ \mathtt{op} \ p'/@B$$

$$Axis ::= \ \epsilon \mid \triangledown \mid \triangledown^* \mid \ \vartriangle \ \mid \ \vartriangle^* \ \mid \triangleright \mid \triangleright^* \ \mid \triangleleft \mid \triangleleft^*$$

où a appartient à l'ensemble $T \cup \{\star\}$, A et B sont des noms d'attributs et $\mathtt{op}$ est égal à $=$ ou $\neq$.

Sémantique. Pour un nœud n d'un document t, t *satisfait* p au nœud n si et seulement s'il existe un nœud n' tel que $t \models p(n, n')$.

La sémantique des requêtes binaires $t \models p(n, n')$ et des requêtes unaires $t \models q(n)$ est définie par induction de la facon suivante :

1. si $p = Axis :: a$, le nœud n' est étiqueté[1] par a si $a \in T$ et
   (a) si $Axis = \epsilon$ alors $n = n'$;
   (b) si $Axis = \triangledown$ alors n' est un enfant de n ;
   (c) si $Axis = \triangledown^*$ alors n' appartient au sous-arbre de racine n ;
   (d) si $Axis = \vartriangle$ alors n' est le père du nœud n ;
   (e) si $Axis = \vartriangle^*$ alors n' est un ancêtre du nœud n ;
   (f) si $Axis = \triangleright$ alors n' est le voisin suivant n ;

[1] Si a est le symbole "wildcard" $\star$, on n'impose aucune contrainte sur l'étiquette du nœud n'.

(g) si $Axis = \triangleright^*$ alors n' est un voisin à droite du nœud n ;

(h) si $Axis = \triangleleft$ alors n' est le voisin précédent le nœud n ;

(i) si $Axis = \triangleleft^*$ alors n' est un voisin à gauche du nœud n ;

2. si $p = p_1/p_2$ alors il existe un nœud n_2 de T tel que $t \models p_1(n, n_2) \wedge p_2(n_2, n')$;
3. si $p = p_1 \cup p_2$ alors $t \models p_1(n, n')$ ou $t \models p_2(n, n')$;
4. si $p = p_1[q]$ alors $t \models p_1(n, n')$ et $t \models q(n')$ où la satisfaction de q est définie de la manière suivante :

   (a) si $q = p'$ alors $t \models \exists n_1 p'(n', n_1)$;

   (b) si $q = p/@A\ \texttt{op}\ p'/@B$: alors $t \models \exists n_1,\ n_2\ (p(n', n_1) \wedge p'(n', n_2) \wedge n_1.A\ \texttt{op}\ n_2.B$;

   (c) si $q = q_1 \wedge q_2$ alors $t \models (q_1(n')$ et $t \models q_2(n'))$;

   (d) si $q = q_1 \vee q_2$ alors $t \models (q_1(n')$ ou $t \models q_2(n'))$;

   (e) si $q = \neg q_1$ alors $t \not\models q_1(n')$.

Des problèmes d'analyse statique

Si on dispose de formalismes pour définir des schémas et de langages pour extraire de l'information, on va vouloir tester si un document satisfait les contraintes décrites par un schéma ou encore calculer l'ensemble des éléments résultant de l'évaluation d'une requête de ces langages. On étudiera ces questions dans la seconde partie de la thèse en considérant le problème de Model-Checking pour le formalisme étudié.

Dans cette partie, on s'intéresse à des problèmes d'analyse statique. Par exemple, lorsqu'on définit un schéma, on peut poser la question de la cohérence de ce schéma : "est-ce qu'il existera vraiment des documents respectant ces contraintes ?". Si on veut regrouper des données stockées suivant différents schémas, on se pose la question d'interopérabilité entre schémas : "Est-ce que mon schéma de données respecte les contraintes imposées par le schéma de mon voisin ?". Un dernier exemple est l'optimisation de requête : on peut se demander si deux requêtes sélectionnent les mêmes ensembles de nœuds sachant que les documents vérifient un schéma particulier.

Plus précisément, on parle des problèmes de *cohérence*, *d'inclusion* et *d'équivalence.*

- On dit qu'un schéma est cohérent s'il existe un document qui respecte les contraintes correspondantes. Étant donné un schéma, une requête est cohérente si il existe un document qui vérifie le schéma et tel que la requête séléctionne un ensemble non nul d'éléments sur ce document.
- On dit qu'un schéma $\mathcal{S}$ est inclus dans le schéma $\mathcal{S}'$ (noté $\mathcal{S} \subseteq \mathcal{S}'$) si et seulement si tout document qui vérifie le schéma $\mathcal{S}$ vérifie le schéma $\mathcal{S}'$. Ce problème s'énonce également pour deux ensembles de contraintes $\mathcal{C}$ et $\mathcal{C}'$ en présence d'un schéma $\mathcal{S}$: on note $\mathcal{C} \subseteq_{\mathcal{S}} \mathcal{C}'$ si et seulement si, parmi les documents qui satisfont le schéma $\mathcal{S}$, tout document qui vérifie les contraintes $\mathcal{C}$ vérifie aussi les contraintes $\mathcal{C}'$. (On parle d'implication de contraintes en présence se schéma.)
- On dit que deux schémas (resp. requêtes) sont équivalents si et seulement s'ils définissent le même ensemble de documents (resp. sélectionne le même ensemble d'éléments pour tout document).

Comme pour le problème d'inclusion, la question d'équivalence de requêtes peut se poser relativement à un certain schéma.

D'un point de vue théorique, on peut voir ces problèmes comme des problèmes de satisfaisabilité. Le problème de cohérence est défini comme tel. Un schéma $\mathcal{S}$ n'est pas inclus dans un schéma $\mathcal{S}'$ s'il existe un document qui satisfait $\mathcal{S}$ mais ne satisfait pas $\mathcal{S}'$; c'est-à-dire si le schéma obtenu en considérant $\mathcal{S}$ et la négation de $\mathcal{S}'$ (noté $\mathcal{S} \cap \neg\mathcal{S}'$) est cohérent. L'équivalence peut se voir comme une double inclusion.

Si on fait abstraction des données, on peut exprimer les schémas et les requêtes propres aux documents XML à l'aide de formalismes bien connus que sont les grammaires, les logiques et les automates d'arbres étiquetés sur un alphabet fini. On peut alors répondre aux questions précédentes en utilisant les outils développés dans le cadre de l'étude des arbres finis [Sch07, CDG$^+$07, Lib06]. Ces problèmes sont au moins décidables, même si leur complexité peut-être élevée.

Si par contre on veut tenir compte des données, il n'existe pas de formalisme générique auquel se ramener. De plus ces problèmes deviennent rapidement indécidables. Le problème de satisfaisabilité pour XPath est indécidable dans le cas général. Les problèmes de cohérence et d'inclusion sont indécidables pour des ensembles de contraintes de clé et clé étrangère en présence de schémas simples de type DTD [FL02]. Le problème de cohérence est décidable si on se restreint à des contraintes unaires [FL02].

Notre objectif

Dans ce cadre, on cherche un formalisme générique qui d'une part a une bonne expressivité par rapport aux langages utilisés en pratique et d'autre part dont le problème de satisfaisabilité soit décidable. On a pour cela besoin d'un modèle théorique pour représenter les documents XML, d'un formalisme générique expressif permettant de décrire des schémas et sélectionner des ensembles d'élements dans un tel modèle, et enfin d'outils algorithmiques afin de décider le problème de satisfaisabilité.

On modélise les documents XML à l'aide d'arbres de données. La suite du chapitre est consacrée à la question des formalismes et au problème de satisfaisabilité. L'objectif est d'étendre les logiques et modèles d'automates proposés dans le chapitre précédent sur les mots de données au cas des arbres de données. On verra dans la dernière section les résultats que l'on peut en déduire dans le contexte des problèmes d'analyse statique pour les documents XML.

Les arbres de données : un modèle pour les documents XML.

D'un point de vue théorique si on regarde simplement la structure d'un document XML, on obtient un arbre au sens classique du terme où les étiquettes sont les tags.

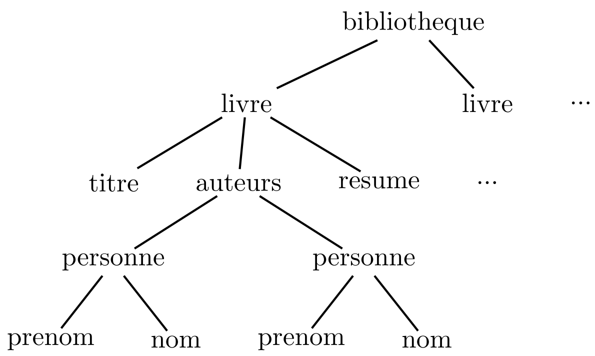


FIG. 3.4 – La structure d'arbre du document

Si on veut tenir compte de tout le document et en particulier des données qui sont a priori quelconques (texte et valeur d'attribut) on va utiliser un modèle d'arbre plus général : les arbres de données. On utilise pour cela un alphabet fini $\Sigma = T \uplus \mathit{Attr}$ constitué d'un ensemble fini de tags T et des noms des attributs Attr, et un ensemble de données $\mathcal{D}$ a priori infini.

> Un **arbre de données** sur l'alphabet Σ est un arbre ordonné d'arité non bornée bi-étiqueté par l'alphabet Σ et l'ensemble de données $\mathcal{D}$. Plus précisément, chaque nœud v est étiqueté par une *lettre* $v.l \in \Sigma$ et une *donnée* $v.d \in \mathcal{D}$.

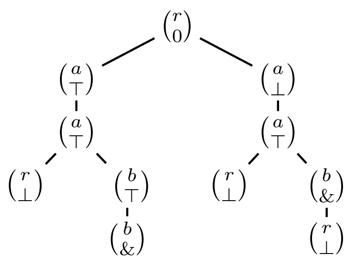


FIG. 3.5 – Un arbre de données

D'un document XML à un arbre de données. Un façon naturelle de voir les documents comme un arbre de données est d'associer, en plus du tag, la donnée au nœud représentant l'élément correspondant. La Figure 3.6 donne ainsi un exemple de codage du document présenté dans l'exemple de bibliothèque. On associe au nœud `titre`, fils du premier nœud `livre`, la donnée `"Le livre qui rend fou"` correspondant au titre du livre. Le nœud `isbn` contient quant à lui la valeur de l'attribut correspondant `210003202`.

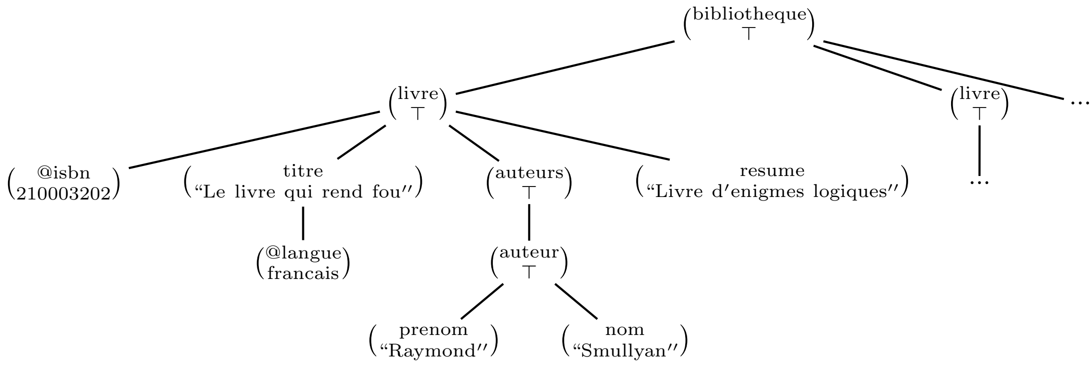


FIG. 3.6 – Un document XML vu comme un arbre de données

On remarque que certains nœuds portent la valeur $\top$, c'est une façon de marquer qu'aucune valeur de données n'est associée dans le document. On pourrait imaginer un autre codage qui utilise ces nœuds pour stocker des informations supplémentaires ou redondantes. Dans l'exemple précédent, on pourrait imaginer ajouter à tous les nœuds `livre` la donnée de l'attribut `isbn` correspondant car ce numéro constitue une imformation importante pour le livre. Dans ce codage modifié, on peut alors imaginer que l'on va supprimer les nœuds `isbn` puisque cette information est redondante ; cependant si on veut ensuite échanger des données à l'aide de ce document, il est peut-être plus prudent de garder la structure générale de l'arbre. On peut imaginer ainsi des variations en fonction des documents considérés. On verra à la fin de ce chapitre que ce type de codages peuvent aider à la décidabilité de certains problèmes pour des langages XML.

3.2 Registres : automates et μ-calcul

On s'intéresse maintenant à étendre au modèle des arbres de données les formalismes proposés précédemment pour décrire des langages de mots de données.

3.2.1 Automates alternants à registres

La notion d'automate alternant à registres s'étend au cas des arbres de données. Les résultats de cette section proviennent des travaux présentés dans [JL07] et [JL08].

Dans la suite on s'intéresse à la classe $ATRA_1(\triangleright, \triangledown)$ que l'on peut voir comme l'extension aux arbres de la classe ARA_1. Intuitivement ces automates sont des automates cheminants[2] alternants munis d'un registre dont la tête de lecture ne peut se déplacer que vers le bas (dans un enfant) et la droite (vers le frère droit) dans l'arbre. Le déplacement de la tête de lecture selon les axes $\triangleright$ et$\triangledown$ correspond en quelque sorte à l'unidirectionalité des automates sur les mots.

[2] Pour plus de détails sur les automates cheminants, on revoie le lecteur au tutorial de Bojanczyk [Boj08].

Définition 3.5 Un automate à un registre $\mathcal{A}$ est un tuple $(Q, q_0, Q_f, \mu_0, \delta)$ où :
- Q est l'ensemble fini des états, q_0 est l'état initial et Q_f l'ensemble des états finaux,
- $\mu_0 \in \mathcal{D}$ est la valuation initiale du registre.
- δ est l'ensemble fini des transitions.

Une transition de l'automate est soit un branchement existentiel $(q, a, \theta) \to q' \vee q"$ soit un branchement universel $(q, a, \theta) \to q' \vee q"$, soit une transition classique $(q, a, \theta) \to (Axis, q', r)$ où $q, q', q" \in Q$, $a \in \Sigma$, $r \in \{\#, 1\}$, $\theta \in \{\sim, \nsim\}$ et $Axis \in \{\triangleright, \triangledown\}$.

Comme dans le cas des mots, on peut définir une configuration acceptante comme une configuration dont l'état associé est final [3].

Cette classe d'automate présente de bonnes propriétés de clôture :

Proposition 3.6 ([JL08]) La classe $ATRA_1(\triangleright, \triangledown)$ est close par union, intersection et négation.

Le résultat principal est :

Théorème 3.7 ([JL08]) *Le problème du vide des automates $ATRA_1(\triangleright, \triangledown)$ est décidable mais non primitif récursif.*

Remarque 3.8 Comme dans le cas des mots on peut étendre la définition d'automate alternant en autorisant plusieurs registres ou encore en autorisant la tête de lecture à se déplacer selon les axes $\vartriangle$ et $\triangleleft$.
Toutes ces extensions entraînent l'indécidabilité du problème du vide comme dans le cas des mots. Le vide des automates alternants à 2 registres est indécidable dans le cas restreint des mots [DL06]. Autoriser le déplacement de la tête de lecture suivant $\vartriangle$ ou $\triangleleft$ permet de simuler les automates bidirectionnels à registres sur les mots de données ; le vide de ces automates est indécidable [Dav04].

Les preuves de ces résultats utilisent un modèle étendu aux arbres des automates à compteurs avec erreur d'incrémentation.

Le cas des arbres infinis

Comme dans le cas des mots, on peut étendre le modèle des $ATRA_1(\triangleright, \triangledown)$ aux arbres infinis. On rajoute une fonction de rang qui associe à chaque état un entier. On considère des automates faibles et la définition d'acceptance pour une exécution infinie est alors une condition de parité sur le rang de ses états.

Le vide des automates de la classe $ATRA_1(\triangleright, \triangledown)$ est indécidable mais on obtient la décidabilité en imposant une restriction syntaxique sur le rang des états des automates. Cette condition est appelée *safety.* Comme dans le cas des mots la classe safety-$ATRA_1(\triangleright, \triangledown)$ est close par union, intersection mais pas par complément. La complexité du vide est cependant bien plus importante que dans le cas des mots car la borne inférieure obtenue est non-primitive récursive [JL08] alors que le problème est ExpSpace dans le cas des mots.

[3] Dans les travaux de Lazić et Jurdziński, les automates sont définis dans un cadre plus général et la condition d'acceptance est donnée comme un condition de parité sur les états. Dans le cas présent, on peut aussi définir cette condition d'acceptance à l'aide de l'ensemble Q_f. Cela permet d'alléger la définition dont le but est principalement de donner au lecteur l'idée du fonctionnement de ces automates. Un lecteur curieux peut trouver la définition originale dans [JL07].

3.2.2 Logiques et registres

Comme dans le cas des mots, on peut définir des formalismes logiques sur les arbres de données en utilisant des registres et utiliser les automates alternants pour manipuler les formules.

Le μ-calcul avec registres

Dans [JL07], R. Lazić et M. Jurdziński présentent une extension du μ-calcul par des registres et l'opérateur $\downarrow$.

Définition 3.9 Les formules du μ-calcul sont définies par la grammaire suivante :

$$\begin{array}{rl} \phi ::= & Z \mid a \mid \overline{a} \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid down \mid up \mid right \mid left \mid \\ & \triangledown\phi \mid \ \vartriangle \phi \mid \vartriangleright \phi \mid \vartriangleleft \phi \mid \ \downarrow_r \phi \mid \ \uparrow_r \ \mid \ \not\uparrow_r \ \mid \overline{down} \mid \overline{up} \mid \overline{right} \mid \overline{left} \\ & \mu Z\phi(Z) \mid \nu Z\phi(Z) \text{ avec } Z \text{ variable libre de } \phi \end{array}$$

où $a \in \Sigma$, Z est une variable et μ et ν sont les opérateurs de plus grand et plus petit point fixe.

L'ensemble $free(\phi)$ des variables libres d'une formule ϕ sont définies par induction de la manière usuelle :
- $free(\phi) = \emptyset$ si ϕ est une formule atomique $a, \bar{a}, down, \overline{down} \dots$;
- $free(Z) = \{Z\}$;
- $free(\phi \vee \psi) = free(\phi \wedge \psi) = free(\phi) \cup free(\psi)$;
- $free(O\phi) = free(\phi)$ pour $O \in \{\triangledown, \vartriangle, \vartriangleright, \vartriangleleft\}$;
- $free(\mu Z\phi(Z)) = free(\nu Z\phi(Z) = free(\phi)\backslash\{Z\}$.

Sémantique : Soit t un arbre des données et S l'ensemble des nœuds de l'arbre. Si ϕ est une formule, λ une valuation des variables libres de la formule et ρ une valuation des registres, on note $\|\phi\|^t_{\lambda,\rho}$ l'ensemble des nœuds où la formule est vraie. La sémantique des formules est définie inductivement de la manière suivante[4] :

$$\begin{array}{lcl}
\|Z\|^t_{\lambda,\rho} & = & \lambda(Z); \\
\|a\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds étiquetés } a\,; \\
\|down\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds qui ne sont pas une feuille}\,; \\
\|up\|^t_{\lambda,\rho} & = & \text{tous les nœuds sauf la racine de } t\,; \\
\|right\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds qui ont un frère à droite}\,; \\
\|left\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds qui ont un frère à gauche}\,; \\
\|\triangledown\phi\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds dont le premier fils existe et appartient à } \|\phi\|^t_{\lambda,\rho}\,; \\
\| \vartriangle \phi\|^t_{\lambda,\rho} & = & \text{l'ensemble de nœuds dont le père existe et appartient à } \|\phi\|^t_{\lambda,\rho}\,; \\
\| \vartriangleright \phi\|^t_{\lambda,\rho} & = & \text{l'ensemble de nœuds qui ont un frère droit qui appartient à } \|\phi\|^t_{\lambda,\rho}\,; \\
\| \vartriangleleft \phi\|^t_{\lambda,\rho} & = & \text{l'ensemble de nœuds qui ont un frère gauche qui appartient à } \|\phi\|^t_{\lambda,\rho}\,; \\
\| \downarrow_r \phi\|^t_{\lambda,\rho} & = & \text{l'ensemble des nœuds } v \text{ qui appartiennent à } \|\phi\|^t_{\lambda,\rho'} \text{ avec } \rho' = \rho_{[r \leftarrow d']} \\
& & \text{où } d' \text{ est la donnée du nœud } v\,; \\
\| \uparrow_r \|^t_{\lambda,\rho} & = & \text{l'ensemble de nœuds qui sont dans la classe correspondant à } \rho(r);
\end{array}$$

[4]On omet volontairement les cas duaux des formules atomiques dont la sémantique se définit naturellement à partir du reste en considérant les ensembles complémentaires.

$$
\begin{array}{lcl}
\|\psi \vee \phi\|^t_{\lambda,\rho} & = & \|\psi\|^t_{\lambda,\rho} \cup \|\phi\|^t_{\lambda,\rho} \\
\|\psi \wedge \phi\|^t_{\lambda,\rho} & = & \|\psi\|^t_{\lambda,\rho} \cap \|\phi\|^t_{\lambda,\rho} \\
\|\mu Z \phi(Z)\|^t_{\lambda,\rho} & = & \bigcap\{S' \subseteq S \mid \|\phi\|^t_{\lambda',\rho} \subseteq S' \text{ où } \lambda' = \lambda_{[Z \leftarrow S']}\}\,; \\
\|\nu Z \phi(Z)\|^t_{\lambda,\rho} & = & \bigcup\{S' \subseteq S \mid \|\phi\|^t_{\lambda',\rho} \supseteq S' \text{ où } \lambda' = \lambda_{[Z \leftarrow S']}\}\,;
\end{array}
$$

On note $\mu_1^{\downarrow}(\rhd, \triangledown)$ le fragment de μ-calcul utilisant un seul registre et les opérateurs modaux $\rhd$ et $\triangledown$.

Il y a une correspondance entre les fragments du μ-calcul et les automates à registres présentés au début de cette section.

Théorème 3.10 ([JL07]) *(a) Pour toute formule close de $\mu_1^{\downarrow}(\rhd, \triangledown)$, on peut construire (en espace logarithmique) un automate de la classe $ATRA_1(\rhd, \triangledown)$ qui reconnaît le même langage.*

(b) Réciproquement, pour tout automate de la classe $ATRA_1(\rhd, \triangledown)$, on peut construire (en espace logarithmique) une formule close de $\mu_1^{\downarrow}(\rhd, \triangledown)$ qui reconnaît le même langage.

Cette correspondance reste valable pour les fragments du μ calcul autorisant plusieurs registres et d'autres d'opérateurs modaux à condition d'étendre les automates à registres de la même manière. Les résultats suivants sont des conséquences de cette correspondance.

Théorème 3.11 ([JL07])

(a) La satisfaisabilité de $\mu_1^{\downarrow}(\triangledown, \rhd)$ est décidable mais non primitive récursive.

(b) La satisfaisabilité est indécidable pour $\mu_1^{\downarrow}(\triangledown, \vartriangle)$, $\mu_1^{\downarrow}(\rhd, \lhd)$, $\mu_2^{\downarrow}(\triangledown)$ et $\mu_2^{\downarrow}(\rhd)$.

Les arbres infinis

On peut définir le μ-calcul sur les arbres infinis. On considère alors le fragment constitué des formules sans alternation de point fixe. On appelle "safety" le fragment constitué des formules n'utilisant que des plus grand points fixe. Le théorème de correspondance entre le μ-calcul et les automates alternant s'étend au cas des arbres infinis et la propriété de "safety" est préservée. Le problème de satisfaisabilité est donc indécidable dans le cas général et décidable pour le fragment safety-$\mu_1^{\downarrow}(\triangledown, \rhd)$.

3.3 La logique du premier ordre

On s'intéresse maintenant à la version étendue de la logique du premier ordre étudiée dans le chapitre précédent. Les modèles des formules sont maintenant des arbres de données ordonnés. La plupart des résultats de cette section proviennent des travaux présentés dans [BDM$^+$06].

On modifie les formules atomiques afin de tenir compte de la structure d'arbre des modèles considérés.

Définition 3.12 La **logique du premier ordre** sur les arbres de données (notée FO($<, +1, \sim$)) est construite à partir des formules atomiques : $x = y$, $x\nabla^* y$, $x \triangleright^* y$, $x\nabla y$, $x \triangleright y$, $a(x)$ pour $a \in \Sigma$ et $x \sim y$ en utilisant la grammaire suivante :

$$\phi ::= \phi \vee \phi \mid \neg\phi \mid \exists x\phi$$

Sémantique. Le domaine d'interprétation des variables est l'ensemble des nœuds d'un arbre de données. Étant donné un modèle t et une fonction f qui associe à chaque variable un nœud du modèle, la sémantique des formules est définie de la manière suivante :

$t, f \models$	$a(x)$	ssi	$f(x)$ est étiqueté par a dans t
$t, f \models$	$x = y$	ssi	$f(x)$ et $f(y)$ correspondent au même nœud de t
$t, f \models$	$y\nabla x$	ssi	$f(y)$ est le père de $f(x)$ dans t
$t, f \models$	$y\nabla^* x$	ssi	$f(x)$ est un descendant de $f(y)$ dans t
$t, f \models$	$y \triangleright x$	ssi	$f(x)$ est le frère suivant de $f(y)$ dans t
$t, f \models$	$y \triangleright^* x$	ssi	$f(x)$ est un frère de $f(y)$ à droite dans t
$t, f \models$	$x \sim y$	ssi	$f(x)$ et $f(y)$ sont dans la même classe dans t
$t, f \models$	$\varphi_1 \vee \varphi_2$	ssi	$t, f \models \varphi_1$ ou $t, f \models \varphi_2$
$t, f \models$	$\neg\varphi$	ssi	$t, f \not\models \varphi$
$t, f \models$	$\exists x\ \psi(x)$	ssi	il existe f' telle que $f'(y) = f(y)$ pour tout $y \neq x$ et $t, f' \models \psi(x)$

On définit la logique existentielle monadique du second ordre de la même manière que dans le cas des mots.

Définition 3.13 La **logique existentielle monadique du second ordre** (notée EMSO) correspond à l'ensemble des formules de la forme :

$$\exists X_0 \ldots \exists X_k\ \phi(X_0, \ldots, X_k)$$

où les X_i sont des variables monadiques du second ordre et $\phi(X_0, \ldots, X_k)$ est une formule de la logique FO augmentée des formules atomiques $x \in X_i$.

La sémantique est la sémantique usuelle.

Fragments. Dans la suite, on s'intéresse à des fragments de ces logiques. On note FO^k la restriction de FO aux formules utilisant au plus k variables. On désigne par FO($<, +1$) l'ensemble des formules de la logique FO($\sim, <, +1$) n'utilisant pas le prédicat binaire $\sim$ d'égalité de données. De la même manière la logique FO($\sim, +1$) correspond aux formules obtenues sans les prédicats ∇^* et $\triangleright^*$. On note de la même manière les fragments correspondants de EMSO : EMSO^k, EMSO($<, +1$), ...

Voici quelques exemples de propriétés que l'on peut exprimer :

Exemple 3.14 Jusqu'au bout des feuilles. Dans le cas des mots, on pouvait identifier la première et dernière position d'un mot. On montre ici comment distinguer la racine, les feuilles ou encore les premiers et derniers fils.

$$racine(x) ::= \forall y\ (\neg\ y\nabla x)$$

$$feuille(x) ::= \forall y\ (\neg\ x\nabla y)$$

$$premierFils(x) ::= \forall y\ (\neg\ y \triangleright x)$$

$$dernierFils(x) ::= \forall y\ (\neg\ x \triangleright y)$$

Exemple 3.15 L'étiquette a est une clé. On dit que a est une clé de l'arbre t si tous les *nœuds* de t étiquetés par a sont dans une classe différente. La formule suivante assure cette propriété :

$$\phi_{KEY_a} ::= \forall x \forall y \ [a(x) \wedge a(y) \wedge x \sim y] \rightarrow \ x = y$$

Exemple 3.16 Contrainte d'inclusion. On définit la formule $\phi_{a \subseteq b}$ qui assure que toute classe qui contient un a contient aussi un b.

$$\phi_{a \subseteq b} ::= \forall x \ a(x) \rightarrow \exists y \ b(y) \wedge x \sim y$$

3.3.1 Le fragment $\mathrm{FO}^2(\sim, +1)$

Dans le cas des arbres de données, le problème de satisfaisabilité devient encore plus complexe. Si on se restreint au fragment successeur (c'est-à-dire autorisant uniquement les prédicats de navigation locale $\triangledown$ et $\triangleright$) le problème est décidable. On donne ici les résultats relatifs à ce fragment. Pour des preuves détaillées, on réfère le lecteur à la version longue de [BDM$^+$06].

Théorème 3.17 ([BDM$^+$06]) *Le problème de satisfaisabilité pour* $\mathrm{FO}^2(\sim, +1)$ *est décidable sur les arbres de données.*

Le résultat prouvé dans [BDM$^+$06] est en réalité le résultat suivant qui est un peu plus général. Il généralise le résultat du Théorème 2.63 présenté dans le cas des mots de données.

Proposition 3.18 ([BDM$^+$06]) La satisfaisabilité de $\mathrm{EMSO}^2(\sim, +1)$ est décidable sur les arbres de données.

Preuve (Idées). La preuve de ce résultat se fait en trois étapes.

1. Expliciter une forme normale pour les formules.
2. Extraire une classe de modèles particuliers pour cette forme normale.
3. Construire un automate pour reconnaître ces modèles.

La première étape consiste à définir une forme normale de données proche de celle présentée dans la preuve du Théorème 2.63.

La seconde étape consiste à montrer qu'une formule en forme normale de données est satisfaisable si et seulement si elle admet des modèles d'une certaine forme. On montre en particulier comment construire un tel modèle à partir d'un modèle quelconque de la formule.

La dernière étape consiste à montrer que l'ensemble des projections sur l'alphabet fini de ces modèles particuliers peut être décrit à l'aide d'un automate d'arbre à contrainte linéaire (LCTA). Ce modèle d'automate est une version étendue des automates à contrainte linéaire utilisés dans la preuve du Théorème 2.63. Le vide de ces automates est décidable.

On obtient ainsi la décidabilité de la logique. $\square$

Une analyse de la preuve du résultat du Théorème 3.17 donne la borne de complexité suivante. La meilleure borne inférieure connue est celle obtenue pour les mots dans la Proposition 2.64 c'est à dire NExpTime-dur.

Proposition 3.19 ([BDM$^+$06]) La complexité du problème est 3NExpTime

3.3.2 La logique $\mathrm{FO}^2(\sim, <, +1)$ et automates à compteurs

On s'intéresse maintenant à la logique $\mathrm{FO}^2(\sim, <, +1)$. La décidabilité du problème de satisfaisabilité est ouvert. On montre cependant que ce problème, restreint aux arbres binaires, est au moins aussi difficile que le problème du vide pour les automates multicompteurs sur les arbres (appelés TVASS). La décidabilité de ce problème est ouverte depuis plusieurs années et est équivalente à un problème ouvert bien connu de logique linéaire : la décidabilité de la logique linéaire multiplicative exponentielle notée couramment MELL [GGS04].

Les automates à compteurs (TVASS).

Un automate d'arbre à compteurs sur les arbres binaires (appelé TVASS) est un automate d'arbre ascendant muni d'un nombre fini k de compteurs positifs.

Définition 3.20 Formellement, un **TVASS** est représenté par $(Q, k, Q_f, \delta_0, \delta)$ où Q est l'ensemble fini des états, Q_f l'ensemble des états finaux, k est le nombre de compteurs, $\delta \subseteq (Q \times \mathbb{N}^k)^2 \times Q \times \Sigma \times \mathbb{N}^k$ est l'ensemble fini des transitions et $\delta_0 \subseteq Q \times \Sigma \times \mathbb{N}^k$ est l'ensemble fini des transitions initiales. (Les transitions initiales affectent à chaque feuille une valuation initiale des compteurs en fonction de son étiquette.)

Une exécution de l'automate affecte à chaque nœud un état et un vecteur de $\mathbb{N}^k$. Pour une feuille étiquetée a, l'état q et le vecteur $\vec{n}$ doivent respecter la relation de transition δ_0 (ie $(q, a, \vec{n}) \in \delta_0$). Les transitions des autres nœuds sont décrites dans le paragraphe suivant.

Un arbre est accepté si il existe une exécution de l'automate telle que l'état affecté à la racine soit acceptant et tous les compteurs valent 0.

Un exécution de l'automate affecte à chaque nœud de l'arbre un état et une valuation des compteurs en fonction de la relation de transition. Les transitions de l'automates sont définies de la manière suivante : chaque transition dépend de trois vecteurs de $\mathbb{N}^k$ appelés $\vec{c}_0, \vec{c}_1, \vec{c}$, de trois états q_0, q_1, q et d'une lettre $a \in \Sigma$. Supposons que le nœud v soit étiqueté a et que l'exécution de l'automate ait affecté les états q_0, q_1 et les valuations $\vec{x}_0, \vec{x}_1$ à ses deux enfants v_0 et v_1. Si les valeurs des compteurs satisfont $\vec{x}_0 \geq \vec{c}_0$ et $\vec{x}_1 \geq \vec{c}_1$ alors la transition $(q_0, \vec{c}_0, q_1, \vec{c}_1, q, a, \vec{c})$ peut être franchie et l'automate assigne au nœud v l'état q et la valuation $\vec{x}_0 - \vec{c}_0) + (\vec{x}_1 - \vec{c}_1) + \vec{c}$. En d'autres mots, l'automate commence par décrémenter les compteurs associés au fils gauche (resp. droit) par le vecteur $\vec{c}_0$ (resp. $\vec{c}_1$). Si les deux opérations de décrément se passent sans problème (c'est-à-dire qu'aucun décrément ne tente de décrémenter un compteur vide), alors les valeurs des compteurs sont additionnées et incrémentées par $\vec{c}$ afin de constituer la nouvelle valuation des compteurs.

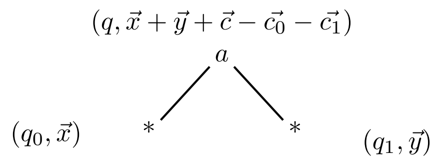


FIG. 3.7 – Transition $(q_0, \vec{c}_0, q_1, \vec{c}_1, q, a, \vec{c})$.

Il est important de remarquer que comme dans le cas des mots ces automates à compteurs ne peuvent pas effectuer des tests à zéro au cours d'une exécution. Dans le cas contraire le vide de ces automates serait évidemment indécidable.

Des TVASS à la logique $\mathrm{FO}^2(\sim, <, +1)$**.**

Théorème 3.21 *Pour tout TVASS* $\mathcal{A}$*, on peut construire une formule* $\varphi_{\mathcal{A}} \in \mathrm{FO}^2(\sim, <, +1)$ *telle que l'ensemble des arbres acceptés par l'automate* $\mathcal{A}$ *est non vide si et seulement si la formule* $\varphi_{\mathcal{A}}$ *est satisfaisable.*

Preuve. On considère l'automate $\mathcal{A} = (Q, Q_f, \delta_0, \delta)$.

On construit une formule $\varphi_{\mathcal{A}}$ de $\mathrm{FO}^2(\sim, <, +1)$ telle que les modèles de cette formule sont exactement le codage des éxécutions acceptantes de $\mathcal{A}$.

Codage d'une exécution. On considère un arbre binaire T étiqueté sur Σ et une exécution ρ de l'automate $\mathcal{A}$ sur T. A partir de l'arbre T on construit un arbre de données T_ρ en ajoutant des nœuds et des valeurs de données pour représenter l'exécution ρ.

L'idée est la même que dans la preuve du Théorème 2.58, on utilise un alphabet étendu pour représenter les états et compteurs de l'automate $\Sigma' = \Sigma \cup Q \cup \{I_j, D_j | j = 1 \dots k\}$ (les nouvelles lettres I_i et D_i représentent respectivement l'incrémentation ou la décrémentation du compteur correspondant).

On commence par décrire la structure finie de l'arbre puis on décrit les valeurs de données.
L'exécution ρ associe à chaque nœud de T un état de $\mathcal{A}$. Entre chaque de T étiqueté Σ et son père, on ajoute un nœud étiqueté par l'état correspondant. Afin de représenter l'état associé à la racine de T, on ajoute un nœud étiqueté par Q qui devient la nouvelle racine de l'arbre. On ajoute maintenant des nœuds afin de représenter les opérations sur les compteurs. On considère un nœud n de T étiqueté a et ses deux fils n_0 et n_1. On appelle q, q_0 et q_1 les états associés à ces nœuds au cours de l'exécution ρ et la transition correspondante $(q_0, (r_1, \cdots, r_k), q_1, (r'_1, \cdots, r'_k), q, a, (z_1, \cdots, z_k)) \in \delta$. Afin de représenter les incrémenations de compteurs, on ajoute des nœuds sur la branche reliant le nœud étiqueté a correspondant à n et son père étiqueté q. Pour tout $i = 1, \dots, k$ on ajoute z_i *nœuds* étiquetés I_i de telle sorte que le chemin de q à a soit étiqueté $I_1^{z_1} \dots I_k^{z_k}$. De la même manière, on représente les décrémentations de compteurs en ajoutant pour tout i r_i *nœuds* étiquetés D_i dans la branche reliant q_0 à a et r'_i *nœuds* étiquetés D_i dans la branche reliant q_1 à a. Le chemin de a à q_0 est ainsi étiqueté $D_1^{r_1} \dots D_k^{r_k}$ et celui de a à q_1 $D_1^{r'_1} \dots D_k^{r'_k}$ La Figure 3.8 donne un exemple.
Les valeurs de données sont utilisées comme dans la preuve du Théorème 2.58 pour vérifier la valeur des compteurs au cours de l'exécution. Les valeurs de données associées aux nœuds étiquetés par $\Sigma \cup Q$ n'ont pas d'importance, on impose que ces nœuds sont tous dans des classes différentes. Toute classe contenant une position étiquetée I_i ou D_i contient exactement deux positions étiquetées respectivement par D_i et I_i pour le même i et la position étiquetée I_i est un ancêtre de la position D_i correspondante. Cette structure de données assure qu'aucun compteur nul n'est décrémenté au cours de l'exécution.

On obtient ainsi un codage T_ρ de l'exécution ρ sur l'arbre T.

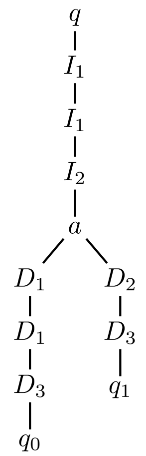


FIG. 3.8 – Codage de la transition $(q_0, (2,0,1), q_1, (0,1,1), q, a, (2,1,0))$.

La formule. On peut construire une formule $\mathrm{FO}^2(\sim, <, +1)$ qui reconnait exactement les codages des exécutions acceptantes de $\mathcal{A}$.
On peut décrire la structure générale de l'arbre à l'aide d'une formule $\mathrm{FO}^2(+1)$ qui assure les propriétés suivantes :

- Un nœud étiqueté par Σ est une feuille ou possède deux enfants étiquetés par $Q \cup \{D_i | i = 1 \dots k\}$ et son père est étiqueté par $Q \cup \{I_i | i = 1 \dots k\}$.
- Un nœud étiqueté par Q a un enfant étiqueté par $\{I_i | i = 1 \dots k\} \cup \Sigma$. S'il ne s'agit pas de la racine de l'arbre son père est étiqueté $\Sigma \cup \{D_i | i = 1 \dots k\}$.
- Un nœud étiqueté par I_j a un enfant étiqueté par $\{I_i | i = j \dots k\} \cup \Sigma$ et son père est étiqueté $Q \cup \{I_i | i = 1 \dots j\}$.
- Un nœud étiqueté par D_j a un enfant étiqueté par $\{D_i | i = j \dots k\} \cup Q$ et son père est étiqueté $\Sigma \cup \{D_i | i = 1 \dots j\}$.

Toujours à l'aide d'une formule $\mathrm{FO}^2(+1)$, on peut vérifier que l'étiquetage des feuilles et de la racine correspond respecte les conditions initiales et d'acceptance de l'automate et que l'étiquetage des nœuds interne respecte les transitions de l'automate :

- La racine est étiquetée par un état final.
- Le chemin ascendant à partir d'une feuille étiqueté $a \in \Sigma$ commence par $I_k^{z_k} \dots I_1^{z_1} q$ où $(q, a, (z_1, \dots z_k)) \in \delta_0$.
- Pour tout nœud interne étiqueté $a \in \Sigma$, il existe une transition de la forme $(q_0, (r_1, \cdots, r_k), q_1, (r'_1, \cdots, r'_k), q, a, (z_1, \cdots, z_k)) \in \delta$ telle que le chemin ascendant à partir de ce nœud commence par $I_k^{z_k} \dots I_1^{z_1} q$, et on peut descendre dans l'arbre(en suivant la relation père/fils) par un chemins étiquetés par $D_1^{r_1} \dots D_k^{r_k} q_0$, mais aussi par un chemin étiqueté[5] par $D_1^{r'_1} \dots D_k^{r'_k} q_1$.

La structure de donnée de l'arbre est contrainte par une formule de $\mathrm{FO}^2(\sim, <)$ construite en utilisant des formules proches de celles présentées dans les Exemples 3.15 et 3.16 □

La décidabilité de la logique $\mathrm{FO}^2(\sim, <, +1)$ reste donc un problème ouvert que l'on sait maintenant difficile.

[5] On peut distinguer les deux chemins à l'aide du prédicat $\triangleright$.

3.3.3 Comparaison avec le μ-calcul et la logique $S\mathrm{CTL}_1^{\downarrow}$

On considère le fragment décidable $\mathrm{FO}^2(\sim,+1)$ et le fragment décidable $\mu_1^{\downarrow}(\triangledown,\triangleright)$. Ces deux logiques sont incomparables en terme d'expressivité. D'un côté, les formules de $\mu_1^{\downarrow}(\triangledown,\triangleright)$ ne peuvent exprimer les propriétés de clé[6] qui sont naturellement exprimables par une formule $\mathrm{FO}^2(\sim,+1)$. Et en même temps, les formules $\mathrm{FO}^2(\sim,+1)$ ne peuvent exprimer la propriété suivante : pour tout nœud de l'arbre, aucun de ses descendants ne contient la même valeur de données que lui.

La traduction classique des formules CTL en systèmes du μ-calcul s'étend aux registres et à l'opérateur $\downarrow$. On note $S\mathrm{CTL}_1^{\downarrow}(+1,<)$ le fragment de CTL avec un registre et l'opérateur $\downarrow$ défini comme suit.

- Les formules atomiques sont $\top$, $\uparrow$, a pour $a \in \Sigma$.
- On autorise les opérateurs booléens $\neg$, $\wedge$ et les opérateurs temporels $\mathtt{QL}$ pour $\mathtt{Q} \in \{\triangledown\mathtt{E}, \vartriangle, \triangleright, \triangleleft\}$ et $\mathtt{L} \in \{\mathtt{X}, \mathtt{XXF}\}$.
- Toute occurrence d'un opérateur temporel est précédé de $\downarrow$

La sémantique est la sémantique usuelle, l'opérateur Q se réfère au déplacement dans l'arbre ($\mathtt{E}$ désignant l'existence d'un chemin à partir d'un nœud courant). On réfère le lecteur à [JL07] pour plus de détails.

Proposition 3.22 La logique $S\mathrm{CTL}_1^{\downarrow}(+1,<)$ est incluse dans $\mu_1^{\downarrow}(\triangledown,\vartriangle,\triangleright,\triangleleft)$.

On distingue deux fragments de cette logique :

- On note $S\mathrm{CTL}_1^{\downarrow}(<)$, le fragment correspondant aux formules restreintes aux opérateurs temporels $\triangledown\mathtt{EXF}$, $\vartriangle\mathtt{XF}$, $\triangleright\mathtt{XF}$, $\triangleleft\mathtt{XF}$.
- On note $S\mathrm{CTL}_1^{\downarrow}(+1)$, le fragment correspondant aux formules restreintes aux opérateurs temporels $\triangledown\mathtt{EX}$, $\vartriangle\mathtt{X}$, $\triangleright\mathtt{X}$, $\triangleleft\mathtt{X}$ et $\mathtt{EXXF}$.
  Où $\mathtt{EXXF}\phi$ est équivalent à $(\triangledown\mathtt{EXXF}\phi) \vee (\vartriangle\mathtt{XXF}\phi) \vee (\triangleright\mathtt{EXXF}\phi) \vee (\triangleleft\mathtt{EXXF}\phi)$

Théorème 3.23 ([JL08]) *Soit* $\mathcal{O} \subseteq \{+1,<\}$

(a) Pour toute formule $\mathrm{FO}^2(\sim,\mathcal{O})$*, on peut construire en espace polynomial une formule de* $SCTL_1^{\downarrow}(\mathcal{O})$ *qui reconnaît le même langage.*

(b) Pour toute formule de $SCTL_1^{\downarrow}(\mathcal{O})$*, on peut construire en temps logarithmique une formule de* $\mathrm{FO}^2(\sim,\mathcal{O})$ *qui reconnaît le même langage.*

Le Théorème 3.17 permet d'obtenir la décidabilité d'un fragment de $\mu_1^{\downarrow}(\triangledown,\vartriangle,\triangleright,\triangleleft)$ dont le problème de satisfaisabilité général est indécidable (voir Théorème 3.11).

Corollaire 3.24 La satisfaisabilité de $S\mathrm{CTL}_1^{\downarrow}(+1)$ est 3NExpTime.

3.4 Applications à XML

Comme on l'a expliqué au début de ce chapitre, on peut voir un document XML comme un arbre de données étiqueté par l'alphabet $\Sigma = T \uplus Attr$, T correspondant aux tags et $Attr$ aux noms des attributs. Une problématique importante liée au langage XML est d'avoir des formalismes expressifs et décidables pour définir des schémas ou encore des langages de requête. Dans cette partie,

[6] Toutes les valeurs de données sont différentes.

on s'intéresse à l'expressivité des logiques et modèles d'automates proposés dans les deux sections précédentes par rapport à cette problématique.

Les résultats de décidabilité obtenus entraînent la décidabilité du problème de satisfaisabilité pour tout schéma ou requête exprimable par une formule $\mathrm{EMSO}^2(\sim,+1)$ ou $\mu_1^{\downarrow}(\triangleright,\triangledown)$, ou encore un automate $ATRA_1$.

3.4.1 Schéma et implication de contraintes

Si on fait abstraction des données, on remarque en particulier que la logique $\mathrm{EMSO}^2(+1)$ permet d'exprimer n'importe quel langage régulier d'arbre (Proposition 3.18). De la même manière, on peut facilement simuler un automate d'arbre descendant à l'aide d'un automate $ATRA_1$ sans utiliser le registre. L'idée est la suivante : à partir d'un nœud, l'automate alternant parcourt l'ensemble des enfants de gauche à droite en utilisant le branchement universel pour continuer à simuler l'exécution de l'automate en descendant plus en profondeur dans l'arbre.

Mais tout l'intérêt des formalismes précédents est de pouvoir parler des données. Les résultats de décidabilité de la satisfaisabilité (resp. du vide) de la logique $\mathrm{EMSO}^2(\sim,+1)$, du fragment de μ-calcul $\mu_1^{\downarrow}(\triangleright,\triangledown)$, ou encore des automates $ATRA_1$ se traduisent de la manière suivante :

Proposition 3.25 Le problème de cohérence est décidable pour n'importe quel schéma exprimable dans un des formalismes suivant : $\mathrm{EMSO}^2(\sim,+1)$, $\mu_1^{\downarrow}(\triangleright,\triangledown)$, $ATRA_1$.

Si on s'intéresse aux *contraintes d'intégrité* présentées au début du chapitre, on a le résultat suivant :

Proposition 3.26 $\mathrm{FO}^2(\sim,+1)$ permet d'exprimer des contraintes de type clé unaire et clé étrangère unaire.

Les propriétés de clés ne semblent pas pouvoir s'exprimer à l'aide d'un automate $ATRA_1$. L'idée est qu'à partir de déplacements vers le bas $\triangledown$ et vers la droite $\triangleright$, il n'est pas possible de construire un chemin qui passe par le fils le plus à droite de la racine et la feuille la plus à gauche de l'arbre[7]. Un automate $ATRA(\triangledown,\triangleright)$ ne semble donc pas pouvoir distinguer un arbre dans lequel ces deux nœuds sont dans la même classe d'un arbre dans lequels ils sont dans des classes différentes.

Le résultat de décidabilité pour la logique entraîne la décidabilité du problème d'implication pour ces contraintes. Le résultat était déjà connu : dans [AFL05], les auteurs montrent que le problème en l'absence de schéma est polynomial.

Les langages réguliers d'arbres peuvent être caractérisés par des formules $\mathrm{EMSO}^2(+1)$. Les Propositions 3.26 et 3.18 entraînent le résultat suivant :

Proposition 3.27 Le problème d'implication en présence de schéma est décidable pour des contraintes d'intégrité de type clé ou inclusion ou clé étrangère unaire.

De manière plus générale, ce résultat est vrai pour toute contrainte d'intégrité exprimable à l'aide d'une formule $\mathrm{FO}^2(\sim,+1)$ et n'importe quel schéma exprimable dans $\mathrm{EMSO}^2(\sim,+1)$.

[7]En supposant bien-sûr que la racine a au moins deux enfants.

3.4.2 Le langage XPath

On s'intéresse maintenant au langage navigationnel XPath. Le problème de satisfaisabilité en présence de schéma est décidable pour toute requête et tout schéma que l'on peut définir dans $\mathrm{EMSO}^2(\sim,+1)$, $\mu_1^{\downarrow}(\triangleright,\triangledown)$ ou $ATRA_1(\triangleright,\triangledown)$.

Les liens entre XPath et la logique FO^2 sont nombreux même sans valeurs de données. On commence par rappeler ces liens et montrer que la logique $\mathrm{FO}^2(\sim,<,+1)$ est strictement incluse dans une extension naturelle (mais indécidable) d'un fragment de XPath autorisant des tests d'(in)égalité sur les données. On donne ensuite deux exemples de fragments de XPath dont la décidabilité découle des résultats présentés sur les arbres de données ; le fragment "Forward-XPath" dont les requêtes peuvent s'exprimer à l'aide d'automates alternants $ATRA_1(\triangleright,\triangledown)$ et le fragment "LocalDataXPath" dont les requêtes peuvent s'exprimer à l'aide de formule $\mathrm{FO}^2(\sim,+1)$.

XPath et FO^2

Sans valeurs de données la logique $\mathrm{FO}^2(<,+1)$ est équivalente au fragment navigationnel de XPath appelé **Core-XPath** [GKP02]. Le fragment Core-XPath correspond à l'ensemble des expressions de la Définition 3.4 qui ne contiennent pas de test sur les données.

Proposition 3.28 ([Mar05]) Core-Xpath et la logique $\mathrm{FO}^2(<,+1)$ sont équivalents en terme d'expressivité sur les arbres ordonnés d'arité non-bornée.

Le problème de satisfaisabilité pour Core-Xpath a été étudié en détail [GKP02, BFG05, GF05]. Ce problème est décidable même en rajoutant des contraintes de type DTD. Afin d'étendre ce fragment navigationnel, on a considéré le fragment **Core-Data-XPath**. Ce fragment correspond à l'extension de Core-XPath qui autorise des tests d'(in)égalité de données, c'est à dire au fragment présenté dans la Définition 3.4. Malheureusement la satisfaisabilité de Core-Data-XPath est indécidable.

Théorème 3.29 ([GF05]) *La satisfaisabilité de Core-Data-XPath est indécidable.*

Dans [BFG05, GF05], les auteurs présentent différents fragments de Core-Data-XPath pour lesquels le problème de satisfaisabilité est décidable (en présence de différentes sortes de DTD). Cependant les seuls fragments décidables autorisant les tests d'égalité de données ainsi que la négation sont très restreints. Le premier fragment décidable autorise uniquement la navigation verticale locale $\vartriangle$, la négation, les tests sur les données. En particulier ce fragment n'autorise pas l'union d'expressions. L'autre fragment décidable autorisant la négation et les tests sur les données autorise l'union mais restreint la navigation à l'opérateur $\triangledown$.

On commence par montrer l'inclusion stricte de $\mathrm{FO}^2(\sim,<,+1)$ dans le fragment Core-Data-XPath. On présente ensuite un nouveau fragment de XPath appelé LocalDataXPath qui autorise des comparaisons de données et la négation sous des formes restreintes. On montre que les problèmes de satisfaisabilité et d'inclusion se réduisent au problème de satisfaisabilité de $\mathrm{FO}^2(\sim,+1)$. On obtient ainsi un nouveau fragment décidable de XPath avec comparaison de données.

Proposition 3.30 $\mathrm{FO}^2(\sim, <, +1)$ est strictement incluse dans Core-Data-XPath.

Preuve. La preuve de ce résultat est proche de la preuve d'inclusion de $\mathrm{FO}^2(<, +1)$ dans unary-TL sur les mots [EVW02]. L'idée est de transformer une formule $\phi(x)$ en une expression Core-Data-XPath qui s'évalue à `vrai` aux mêmes nœuds que $\phi(x)$. On raisonne pour cela par induction sur la profondeur de quantification de la formule $\phi(x)$.

On considère les formules de la forme $\exists y\ (\tau \wedge \psi(y))$, où τ est une formule qui décrit entièrement l'information relative aux positions x et y (étiquette, valeurs de données et ordre relatif). Par exemple τ peut être de la forme $x \nabla^* y \wedge x \neg \nabla y \wedge a(x) \wedge b(y) \wedge x \sim y$, avec $a, b \in \Sigma$. Toute formule τ se traduit naturellement dans la syntaxe de Core-Data-XPath. La formule précédente se traduit par $\epsilon :: a/@A = \nabla :: \star/\nabla^* :: b/@A$. La taille de l'expression ainsi obtenue est exponentielle dans la taille de la formule de départ $\phi(x)$.

Cette inclusion est stricte. On considère l'expression suivante :

$$[\epsilon :: \star @A = \epsilon :: \star/\nabla^* :: b/\nabla^* :: \star @A]$$

Cette expression évaluée en un nœud n teste si la valeur de l'attribut A de ce nœud est égale à la valeur d'attribut A d'un nœud n' qui est le descendant d'un descendant du nœud n étiqueté par b.

L'existence d'un tel nœud se traduit comme l'existence d'un chemin direct[8] vérifiant la propriété suivante : "Il existe deux positions n et n' dans la même classe telles qu'il existe une position étiquetée par un b entre n et n'".

On montre dans le Lemme 3.31 que cette propriété n'est pas définissable par $\mathrm{FO}^2(\sim, <, +1)$ dans le cas des mots. La même preuve peut être utilisée dans le cas des arbres pour montrer que l'existence d'un chemin vérifiant la propriété n'est pas exprimable par $\mathrm{FO}^2(\sim, <, +1)$. □

On rappelle que la question de la décidabilité de la logique $\mathrm{FO}^2(\sim, <, +1)$ reste ouverte.
On appelle $\mathcal{P}$ la propriété suivante :

> *Il existe deux positions x et y dans la même classe telles qu'il existe une position étiquetée par un b entre x et y.*

Lemme 3.31 La propriété $\mathcal{P}$ n'est pas exprimable dans $\mathrm{FO}^2(\sim, <, +1)$ sur les mots de données.

Preuve. On montre ce résultat en utilisant le *jeu de Ehrenfeucht-Fraïssé*. On présente ici le cas restreint à la logique du premier ordre à deux variables pour plus de détails le lecteur peut se référer à [Lyn97].

Etant donné un entier k et deux mots de données $W_\top$ et $W_\perp$, le jeu d'*Ehrenfeucht-Fraïssé* $\mathbf{EF}^k(W_\top, W_\perp)$ est un jeu à deux joueurs, l'un appelé *Copieur* et l'autre *Saboteur*. L'arêne du jeu est consitituée de deux mots de données : $W_\top$ et $W_\perp$ et quatres jetons p_1, p_1', p_2 et p_2'. Les jetons p_1 et p_2 sont associés à $W_\top$ et les jetons p_1' et p_2' sont associés à $W_\perp$. Une partie se joue en k tours. Chaque tour est consitué d'un coup de Saboteur suivi d'un coup de Copieur. Saboteur choisit un jeton dans l'un de deux arbres et le déplace[9] ; Copieur choisit le jeton

[8] c'est-à-dire le chemin obtenu en suivant la relation père/enfant.

[9] Les jetons sont associés à un mot. Saboteur n'a donc pas le droit de déplacer le jeton dans l'autre mot.

correspondant dans l'autre arbre et le déplace à son tour. Saboteur gagne si à la fin d'un tour les deux paires de jetons (p_1, p_2) et (p'_1, p'_2) ne satisfont pas les mêmes prédicats de la logique (étiquettes, (in)égalité de données, positions respectives selon les prédicats $<$, $+1$). En d'autres termes, Copieur gagne si il peut toujours "copier les mouvements de Saboteur".

> Une propriété n'est pas définissable en $\mathrm{FO}^2(\sim, <, +1)$ ssi quelque soit le nombre de tours k considéré, il existe deux mots $W_\top$ et $W_\perp$ tels que $W_\top$ satisfait $\mathcal{P}$ et $W_\perp$ ne satisfait pas $\mathcal{P}$ et Copieur a une stratégie gagnante pour le jeu $\mathbf{EF}^k(W_\top, W_\perp)$.

Soit k un entier, on commence par décrire les structures $W_\top$ et $W_\perp$ puis on décrit la stratégie gagnante de Copieur pour le jeu $\mathbf{EF}^k(W_\top, W_\perp)$.

Les Structures $W_\top$ et $W_\perp$.

Dans la suite u est le mot $a^n c a^n$, v est le mot $a^n d a^n$ et w est le mot $a^n b a^n$ où $n = 2k$. Les mots $W_\top$ et $W_\perp$ sont définis de la manière suivante :

- la projection sur l'alphabet fini de $W_\top$ est $(uvw)^n uwv (uvw)^n$;
- la projection sur l'alphabet fini de $W_\perp$ est $(uvw)^n uv (uvw)^n$;
- dans les deux mots toutes les positions étiquetées a ou b ou c sont dans des classes différentes.
- dans les deux mots une position étiquetée d est dans la même classe que la position étiquetée c qui la précède.

On peut remarquer que dans les deux structures, deux positions adjacentes sont toujours dans des classes différentes.

Pour des raisons de clareté, on découpe ces mots en fonction des facteurs u, v et w de la projection sur l'alphabet fini. On obtient :

$$W_\top = U_{-n}V_{-n}W_{-n} \dots U_{-1}V_{-1}W_{-1}\ U_0W_0V_0\ U_1V_1W_1 \dots U_nV_nW_n$$

$$W_\perp = U_{-n}V_{-n}W_{-n} \dots U_{-1}V_{-1}W_{-1}\ \ U_0\ V_0\ \ U_1V_1W_1 \dots WnU_nV_nW_n$$

où la projection sur l'alphabet fini des mots de données U_i, V_i et W_i sont respectivement u, v et w. Dans la suite on parle de blocs pour désigner les sous-mots U_i, V_i ou W_i.

Le mot $W_\top$ satisfait la propriété $\mathcal{P}$ grâce aux positions étiquetées c, b et d dans $U_0W_0V_0$. Le mot $W_\perp$ ne satisfait pas $\mathcal{P}$.

Le mot $W_\perp$ peut être vu comme le mot $W_\top$ auquel on aurait enlevé le bloc W_0. Il existe un bijection implicite entre les positions de $W_\perp$ et les positions de $W_\top$ n'appartenant pas à W_0. Dans la suite lorsque l'on dit que deux jetons n'appartenant pas au même mot sont dans la même position, on fait référence à cette bijection. De la même façon, on peut définir des bijections entre les positions des différents blocs. Dans la suite lorsque l'on parle de positions correspondantes entre deux blocs, on fait référence à ces bijections. De plus on parle de blocs sont isomorphes pour désigner deux blocs ayant la même projection sur l'alphabet fini (c'est-à-dire u, v ou w). De plus pour tout $i \in \{0, \dots, n\}$, on appelle *espace* i l'ensemble des blocs B_j pour $j \in \{-i \dots, i\}$ et $B \in \{U, V, W\}$. Et pour tout bloc, on appelle *zone* i l'ensemble des positions $\{1, \dots i\} \cup \{2n+2-i, \dots 2n+1\}$.

On s'intéresse maintenant au jeu $\mathbf{EF}^k(W_\top, W_\perp)$. On appelle *état du jeu* la donnée de la position des jetons sur $W_\top$ et $W_\perp$. On dit qu'un état du jeu est dit *gagnant* (du point de vue de Copieur) si les deux jetons de $W_\perp$ vérifient les mêmes prédicats unaire et binaires de base de la logique FO^2 que ceux que vérifient les jetons du mot $W_\top$. Lors d'un tour, on appelle *jetons fixes* la paire de jetons qui n'est pas déplacée (notée (p, p')) et *jetons mobiles* la paire de jeton déplacée (notée (q, q')).

On montre que Copieur a une stratégie qui assure que chaque paire de jetons (p, p') et (q, q') vérifient une propriété d'invariant $\mathcal{I}\ (i)$ sur l'état du jeu à chaque tour i. Cet invariant nous permet de montrer que la stratégie est gagnante pour Copieur.

Une paire de jetons vérifie l'*invariant* $\mathcal{I}\ (i)$ si et seulement si (0) il existe $j \in \{1, \ldots 2n+1\}$ tel que chacun des deux jetons est à la j ième position du bloc qui le contient et une des propriétés suivantes est vérifiée :

1. Les deux jetons sont dans la même position de $W_\top$ et $W_\bot$.
2. Les jetons sont dans des blocs isomorphes de l'espace i.
   C'est à dire qu'il existe $\ell, \ell' \in \{-i, \ldots, 0, \ldots, i\}$ et $B \in U, V, W$ tels que un des jetons est dans le bloc B_ℓ de $W_\top$ et l'autre dans le bloc $B_{\ell'}$ de $W_\bot$ (avec $\ell' \neq 0$ si $B = W$).
3. Les jetons sont dans deux blocs de l'espace i à une position $j \in \{1, \ldots i\} \cup \{2n+2-i, \ldots 2n+1\}$ (c'est-à-dire de la zone i).

Cet invariant est monotone, deux jetons vérifiant $\mathcal{I}\ (i)$ vérifient automatiquement $\mathcal{I}\ (i+1)$. De plus on peut remarquer que l'invariant garantit que les deux jetons sont étiquetés par le même lettre.

On montre par induction que Copieur a une stratégie qui garantit qu'à chaque tour i, l'état du jeux est gagnant et les paires de jetons (p, p') et (q, q') vérifie $\mathcal{I}\ (i)$.
Tour 0.
On pose à ce tour une premiere paire jetons qui sera la paire fixe du tour suivant. Si Saboteur joue à l'extérieur de W_0 (dans $W_\top$ ou $W_\bot$), Copieur peut jouer au même endroit dans l'autre structure. Si le Saboteur pose son jeton dans W_0, Copieur pose son jeton à la position correspondante du bloc W_1 de $W_\bot$. L'invariant $\mathcal{I}$ est vérifié (la paire de jetons posée vérifie (1) ou (2)). De plus l'état du jeu est gagnant ; on a qu'un seul jeton sur chaque structure, donc seul les étiquettes comptent et elles ont égales.

Tour $i+1$.
On suppose que la paire des jetons fixes vérifie $\mathcal{I}$ (i). On considère plusieurs cas : soit (A) Saboteur pose son jeton dans une position adjacente au jeton fixe de la structure choisie ; soit Saboteur pose son jeton dans un position non adjacente et (B) les jetons fixes vérifient la condition (1) de $\mathcal{I}$ (i), (C) les jetons fixes vérifient la condition (3), ou (D) les jetons fixes vérifient la condition (2). (A) Supposons que Saboteur pose son jeton dans une *position adjacente au jeton fixe* de la structure correspondante. Dans ce cas Copieur fait de même (en respectant bien évidemment l'ordre relatif des jetons).
Il n'est pas difficile de voir que les jetons vérifient $\mathcal{I}$ $(i+1)$. Par induction, les jetons fixes vérifient $\mathcal{I}$ (i) donc $\mathcal{I}$ $(i+1)$. La propriété (0) est immédiate. Dans la plupart des cas, les jetons fixes et les jetons mobiles vérifient la même condition de l'invariant $\mathcal{I}(i+1)$. Cependant des différences peuvent apparaitre si un des jetons de W_0 se trouve à la première ou dernière position de W_0. Par exemple, si les jetons fixes sont tous les deux dans le dernière position du bloc U_0 et que Saboteur joue la positions successeur, le jeton q se retrouve dans le première position de W_0 et le jeton q' dans la première position de V_0 ce qui correspond à la propriété (3) de l'invariant. Un autre exemple : si les jetons fixes p et p' se situent respectivement à la première position de W_0 et W_i (condition (2)) et Saboteur joue le prédécesseur dans une des deux structures, les jetons mobiles q et q' se retrouvent respectivement à la dernière position de U_0 et de V_i (condition (3)).
L'état résultant est gagnant : la stratégie garantie que la position relative des jetons est respectée ; les positions des jetons étant adjacentes, les valeurs de données des jetons sont différentes dans les deux structures ; enfin l'invariant garantit que les conditions d'étiquetage sont respectées.

(B) Supposons maintenant que Saboteur ne joue pas dans une position adjacente et que les jetons fixes sont dans le même position (condition (1)).
Si Saboteur joue dans le bloc W_0, alors Copieur joue dans la position correspondante dans le bloc W_1 ou W_{-1} afin de respecter l'ordre respectif vis à vis des jetons fixes. L'invariant est vérifié car les jetons mobiles vérifient les conditions (0) et (2). L'état est gagnant : par construction la position relative des jetons est respectée ; l'invariant assure les conditions d'étiquetage ; enfin les données des positions des blocs W sont toutes dans des classes différentes ce qui assure les contraintes de données.
Si les jetons fixes sont dans le dernière position de U_0 et Saboteur décide de poser le jeton q dans la première position de V_0. Copieur ne peut pas poser le jeton q' dans la même position sans violer la condition d'adjacente et perdre la partie. La stratégie de Copieur est alors de jouer dans le première position de V_1 (étiquetée par a). Cela assure l'invariant (condition (2)) et l'état est gagnant (tous les a sont dans des classes différentes).
De la même façon si les jetons fixes sont dans la première position de V_0 et Saboteur pose p' dans la dernière position de U_0, Copieur joue dans la dernière position de U_{-1}.
Dans tous les autres cas, Copieur peut jouer dans la même position que Saboteur. L'invariant est respecté puisque les deux paires de jetons vérifient (0) et (1) et l'état est gagnant.

(C) Supposons maintenant que Soiler ne joue pas dans une position adjacente et que les jetons sont dans deux blocs de l'espace i dans une position j de la zone i

(condition (3)). Dans la suite on note ℓ et m les indices des blocs correspondants et on suppose que ℓ correspond à l'indice de la structure dans laquelle Saboteur décide de jouer.
Si Saboteur joue dans une position n'appartenant pas à l'espace $i+1$, Copieur peut jouer dans la même position. L'invariant est vérifié et l'état est gagnant.
Si Saboteur joue dans la zone $i+1$ un bloc de l'espace $i+1$ alors Copieur joue à la même position dans n'importe quel bloc de l'espace $i+1$ en respectant les conditions d'ordre entre les jetons. L'invariant est respecté (condition (3)) et l'état est gagnant.
Si Saboteur joue dans un bloc de l'espace $i+1$ à l'extérieur de la zone $i+1$ alors Copieur joue dans n'importe quel bloc isomorphe de l'espace $i+1$ en respectant les conditions d'ordre entre les jetons. L'invariant est respecté (condition (3)) et l'état est gagnant.

(D) Supposons enfin que Soiler ne joue pas dans une position adjacente et que les jetons fixes sont dans deux blocs isomorphes de l'espace i (condition (2)). Dans la suite on note ℓ et m les indices des blocs correspondants et on suppose que ℓ correspond à l'indice de la structure dans laquelle Saboteur décide de jouer.
Si Saboteur joue dans une position n'appartenant pas à l'espace $i+1$, Copieur peut jouer dans la même position. L'invariant est vérifié et l'état est gagnant.
Si Saboteur joue dans l'espace i en dehors des blocs U_ℓ,V_ℓ et W_ℓ, dans la plupart des cas Copieur peut jouer dans la position correspondante du bloc isomorphe B_{-i-1} ou B_{i+1} afin de respecter l'ordre des jetons. Dans les autres cas [10] les jetons fixes vérifient la condition (3), Copieur joue alors selon la stratégie correspondante.
Si Saboteur joue dans un des blocs U_ℓ,V_ℓ ou W_ℓ , dans la plupart des cas, Copieur peut jouer dans la position correspondante dans les blocs U_m,V_m ou W_m de tel sorte que l'invariant soit vérifié et l'état résultant gagnant. Les cas différents correspondent soit au bloc W_0, soit à la dernière position de U_0 ou première position de V_0. Dans ces cas là, Copieur peut, selon la position du jeton fixe, jouer dans un des blocs correspondant d'indice -1 ou 1 sans perdre et en respectant l'invariant. □

Forward-XPath et $ATRA_1$

On présente ici un fragment de XPath présenté dans [JL08] inspiré par le modèle des automates $ATRA_1$ présenté dans la partie 3.2.

Définition 3.32 Le fragment **forward-XPath** est l'ensemble des requêtes XPath décrites par : ($a \in T$ et $A \in Attr$)

$$p ::= Axis \mid p/p \mid p \cup p \mid p[q]$$
$$q ::= \neg q \mid q \wedge q \mid p \mid a \mid \epsilon@A = \mathtt{s}p@A \mid \epsilon@A \neq \mathtt{s}p@A$$
$$Axis ::= \epsilon \mid \triangledown \mid \triangleright \mid \triangledown^* \mid \triangleright^*$$

Les chemins utilisés dans les tests de valeurs sont définis par

$$\mathtt{s}p ::= \epsilon \mid \triangledown/\mathtt{s}p \mid \triangleright/\mathtt{s}p$$

[10] Par exemple si le jeton p' est dans la dernière position du bloc V_i et que Saboteur place le jeton q dans la première position de W_{i+1}.

Ce fragment de XPath correspond à un fragment dont la navigation restreinte vers l'avant (vers le bas et la droite) et les tests d'(in)égalité sont restreints à des tests entre le nœud courant et un nœud atteignable en un nombre fini de déplacements non récursifs. La sémantique des requêtes est la sémantique standard décrite au début du chapitre.

Théorème 3.33 *Étant donnée une requête de forward-XPath, on peut construire en temps logarithmique un automate* $ATRA_1(\triangledown, \triangleright)$ *équivalent.*

La classe des automates $ATRA_1(\triangledown, \triangleright)$ est close par union. On en déduit donc le théorème suivant dans lequel le mot schéma désigne n'importe quel schéma que l'on peut traduire en un automate $ATRA_1(\triangledown, \triangleright)$.

Théorème 3.34 ([JL07]) *La satisfaisabilité de forward-Xpath en présence de schéma est décidable.*

Le langage LocalDataXPath

On présente maintenant un autre fragment de XPath qui autorise les comparaisons entre les valeurs d'attributs de manière restreinte par rapport à Core-Data-Xpath :

(1) les opérateurs de navigation $\triangledown^*$ et $\triangleright^*$ ne sont pas autorisés,

(2) dans une comparaison de valeurs d'attribut, soit un des deux nœuds est spécifié par un chemin absolu (c'est-à-dire donné à partir de la racine), soit les deux nœuds sont spécifiés par un chemin relatif dont l'expressivité est très restreinte.

Définition 3.35 Le langage **LocalDataXPath** est constitué de l'ensemble des expressions suivantes qui vérifient la propriété syntaxique de *sûreté*.

On définit les expressions par la grammaire suivante où $a \in T$, $A \in Attr$ et $\texttt{eq} \in \{=, \neq\}$:

$$p ::= \texttt{Axis} :: a \mid p/p \mid p[q] \mid /p$$

$$\texttt{Axis} ::= \triangledown \mid \ \vartriangle \ \mid \triangleright \mid \triangleleft \mid \epsilon \mid \boxtimes$$

$$q ::= p \mid q \vee q \mid \neg q \mid \epsilon :: a[q]@A \ \texttt{eq Axis} :: a[q]@A \mid p@A \ \texttt{eq} \ /p@A$$

La propriété de sûreté restreint l'ensemble des expressions de la forme $\epsilon :: a[q]@A \ \texttt{eq Axis} :: b[q']@B$ (où $a, b \in T$; $A, B \in Attr$; eq est $=$ ou $\neq$ et q et q' sont des expressions unaires) apparaissant dans une formule. On dit qu'un tag a est *associé* à un attribut A dans une expression si et seulement si l'expression contient un test de la forme $\epsilon :: a[q]@A \ \texttt{eq Axis} :: b[q']@B$ ou $\epsilon :: b[q]@B \ \texttt{eq Axis} :: a[q']@A$. On dit qu'une expression est *sûre* si tout tag apparaissant dans des tests de la forme $\epsilon :: a[q]@A \ \texttt{eq Axis} :: b[q']@B$ est associé à au plus un attribut.

La sémantique de ces expressions est la sémantique usuelle, on précise uniquement certaines notations ainsi que la sémantique de opérations correspondantes.

L'axe $\boxtimes$ correspond à un déplacement arbitraire dans un nœud différent du document ; il décrit donc les paires de nœuds de la forme (x, y) avec $x \neq y$. Sans ce dernier axe de navigation, le langage serait limité à une navigation locale. Cet axe a donc une grande importance en matière d'expressivité.

Les chemins de la forme $/p$ correspondent aux chemins partant de la racine. On parle dans la suite de chemin absolu pour désigner ce type de chemin et de chemin relatif pour désigner tout chemin qui n'est pas absolu. On peut simuler un chemin absolu à l'aide de l'axe $\boxtimes$ et des tests.

On peut ajouter à la syntaxe le symbole "wildcard" habituellement noté $\star$ et utilisé pour représenter un tag arbitraire. Cela ne change rien à l'expressivité du fragment car $\star$ peut s'exprimer à l'aide de disjonctions.

L'idée de cette restriction syntaxique de "sûreté" est de pouvoir, dans le codage d'un arbre, stocker la valeur d'un attribut directement dans le nœud contenant l'étiquette plutôt que dans un fils étiqueté par le nom de l'attribut.

Exemple 3.36 L'expression suivante sélectionne un nœud étiqueté par le tag `a` si sa valeur d'attribut `B` est la même que celles de tous ses enfants qui sont étiquetés `b` :

$$\neg(\epsilon :: a/@B \neq \triangledown :: b/@B)$$

L'expression suivante est satisfaite par tout nœud étiqueté `a` qui a un fils `b` dont la valeur de l'attribut `C` est la même que la valeur d'attribut `B` de son père :

$$\epsilon :: a@B = \triangledown :: b/@C$$

Ces deux expressions sont sûres si on les considère séparément, mais l'ensemble de ces deux expressions n'est pas "sûre" car l'étiquette `b` est associée à `B` et `C`.

Théorème 3.37 *Les problèmes de satisfaisabilité et d'inclusion sont décidables pour le langage LocalDataXPath.*
Ce résultat est aussi valable en présence d'un schéma constitué d'un langage régulier d'arbre et de contraintes d'intégrité de type clé ou inclusion.

Ce résultat est valable pour les expressions unaires ou binaires.
Preuve. L'idée est de construire à partir d'une expression une formule $FO^2(\sim, +1)$ équivalente. On code les documents XML en utilisant le modèle XPath. Si les expressions ne contiennent pas de test d'(in)égalité de valeurs, il suffit de considérer la transformation standard de Core-XPath dans FO^2 présentée dans [Mar05].

Cela s'étend facilement aux expressions d'(in)égalité contenant au plus une expression de chemin relative. On commence par simuler le chemin relatif puis on devine un autre nœud contenant la même valeur de données (ou au contraire une valeur différente) et on vérifie que ce nœud vérifie bien le chemin absolu donné par l'expression en simulant celui-ci en arrière jusqu'à atteindre la racine. Il est très important ici que le second chemin soit un chemin absolu car les deux variables de la formule sont nécessaires à la navigation. On a donc oublié le nœud de départ lorsque l'on arrive au bout du premier chemin.

Voici un exemple d'expression LocalDataXPath :

$$\triangledown :: \mathtt{a}/\triangledown :: \mathtt{b}@B \;=\; /\triangledown :: \mathtt{c}/\triangleright :: \mathtt{d}@C$$

qui se traduit de la manière suivante en une formule $\mathrm{FO}^2(\sim, +1)$:

$$\begin{aligned}
&\exists y \triangledown(x,y) \wedge a(y) \wedge \\
&\quad \exists x \triangledown(y,x) \wedge b(x) \wedge \\
&\quad\quad \exists y \triangledown(x,y) \wedge B(y) \wedge \\
&\quad\quad\quad \exists x\ x \sim y \wedge C(x) \wedge \\
&\quad\quad\quad\quad \exists y \triangledown(y,x) \wedge d(y) \wedge \\
&\quad\quad\quad\quad\quad \exists x \triangleright (x,y) \wedge c(x) \wedge \\
&\quad\quad\quad\quad\quad\quad \exists y \triangledown(y,x) \wedge \neg \exists x \triangledown(x,y).
\end{aligned}$$

Il reste maintenant à expliquer comment on peut transformer les expressions qui contiennent des (in)égalités entre deux nœuds décrits par des chemins relatifs. La propriété de sûreté impose que pour chaque tag du document au plus un de ses attributs apparaît dans les (in)égalités relatives. On note $Attr_\tau$ l'attribut ainsi associé au tab τ.

Dans le codage d'un arbre de données seules les valeurs de données correspondant à des nœuds attributs ont une signification du point de vue du document XML. On va donc utiliser les valeurs de données des autres nœuds afin d'introduire de la redondance pour pouvoir vérifier l'expression à l'aide d'une formule FO^2.

On associe donc à tout nœud étiqueté par τ la donnée de son fils étiqueté $Attr_\tau$ si il existe. La formule $\mathrm{FO}^2(\sim, +1)$ peut vérifier que la valeur de données stockée dans le nœud élément correspond bien à la valeur de l'attribut. Étant donné cette redondance d'information, on peut exprimer toutes les (in)égalités du fragment à l'aide de formules du premier ordre.

Prenons l'exemple de la sous-expression $\epsilon :: a@C = \triangledown :: b@B$. On considère les arbres de données qui représentent des documents XML et tels que la valeur de données associée à un nœud a est la même que celle de son fils C et la valeur de données associée à un nœud b est la même que celle de son fils B. Sur ces modèles restreints, la sous-expression précédente s'exprime à l'aide de la formule $a(x) \wedge \exists y \triangledown(x,y) \wedge b(y) \wedge x \sim y$.

On peut transformer les autres expressions de la même manière.

La décidabilité du problème de satisfaisabilité en présence d'un schéma[11] est immédiate sachant que l'on peut traduire les langages réguliers et les contraintes dans la logique. $\square$

Remarque 3.38 Si on considère le fragment étendu de LocalDataXPath par tous les axes de navigation $\triangleright^*$, $\triangleleft^*$, $\triangledown^*$ et $\vartriangle^*$ on peut étendre la preuve précédente afin de réduire les problèmes de satisfaisabilité et d'inclusion de ces expressions (en présence de schéma de type langage régulier et contraintes d'intégrité) à la satisfaisabilité de formules $\mathrm{EMSO}^2(\sim, <, +1)$. Malheureusement, on ne sait pas si le problème de satisfaisabilité est décidable pour cette logique.

[11] On entend par schéma un schéma constitué d'un langage régulier d'arbre et de contraintes d'intégrité de type clé ou inclusion.

3.5 Conclusion

Dans ce chapitre, on a étendu les formalismes précédents aux arbres de données. Le modèle des automates alternants à un registre s'étend bien aux arbres de données. Le problème du vide est décidable pour la classe des automates $ATRA_1(\triangledown, \triangleright)$ qui correspond à l'extension aux arbres des automates $1ARA_1$ décrits dans le chapitre précédent. Ce modèle d'automate nous permet de montrer la décidabilité de fragments du μ-calcul avec registres.

La logique FO^2, s'étend elle aussi au cas des arbres de données. Le problème de satisfaisabilité est décidable pour le fragment $\mathrm{FO}^2(\sim, +1)$ et dans la classe de complexité NExpTime. La complexité exacte du problème n'est pas connue, la meilleure borne inféreure étant NExpTime-difficile. La décidabilité du problème de satisfaisabilité pour le fragment $\mathrm{FO}^2(\sim, <, +1)$ reste un problème ouvert. On montre que ce problème est au moins aussi difficile que le problème du vide des TVASS, une extension aux arbres des automates multicompteurs présentés dans le chapitre précédent. En termes d'expressivité, la logique FO^2 et le fragment décidable du μ-calcul à registres sont incomparables.

Les résultats de décidabilité permettent de préciser la frontière de décidabilité de certains problèmes d'analyse statique liés aux documents XML. On obtient la décidabilité du problème de cohérence pour toute classe de schéma exprimable par une formule $\mathrm{FO}^2(\sim, +1)$, ou un automate $ATRA_1(\triangledown, \triangleright)$. Ceci est intéressant car ces deux formalismes permettent ainsi de combiner n'importe quel langage régulier d'arbre avec des contraintes sur les données. En particulier les propriétés de clé et clé étrangère unaire sont exprimables dans la logique $\mathrm{FO}^2(\sim, +1)$. De la même manière, on obtient la décidabilité du problème d'inclusion en présence de schéma pour toutes contraintes et schémas exprimables dans $\mathrm{FO}^2(\sim, +1)$ ou par un automate $ATRA_1(\triangleright, \triangledown)$. Ce résultat étend le résultat donné dans [AFL05, FL02] qui était restreint aux clés et clés étrangères et aux schémas de type DTD.

Un autre résultat intéressant découle de la comparaison de ces formalismes avec le langage XPath. Le problème de satisfaisabilité en présence de schéma est décidable pour tout fragment XPath et schéma exprimables par une formule $\mathrm{FO}^2(\sim, +1)$, ou un automate $ATRA_1(\triangledown, \triangleright)$. On identifie, en particulier, deux fragments de XPath permettant des comparaisons de données restreintes. Le fragment forward-XPath est un fragment dont les axes de navigation sont restreints à la navigation descendante et vers la droite. Pour toute expression de ce fragment, on peut construire un automate $ATRA_1(\triangledown, \triangleright)$ qui reconnaît le même langage. Dans le fragment *LocalDataXPath* la navigation correspond aux axes locaux mais est augmentée par un nouvel axe $\boxtimes$ qui permet de se déplacer dans une position quelconque du document. On impose des restrictions syntaxiques sur les tests de données afin de pouvoir réduire la satisfaisabilité et le problème d'implication pour les expressions *LocalDataXPath* au problème de satisfaisabilité de la logique $\mathrm{FO}^2(\sim, +1)$. On obtient ainsi un nouveau fragment décidable de XPath avec comparaisons de données.

Il est intéressant de remarquer que les formalismes décrits permettent de regrouper les requêtes XPath et les schémas et contraintes d'intégrité. Ces formalismes étaient jusqu'ici étudiés en général séparément.

On regrettera cependant la complexité extrême des problèmes considérés. Le problème du vide pour les automates $ATRA_1(\triangleright, \triangledown)$ est non primitif récursif et indécidable pour toutes les extensions naturelles considérés. Le problème de satisfaisabilité de la logique $\mathrm{FO}^2(\sim, +1)$ est 3NExpTime et NExpTime-dur et le problème pour le fragment plus général $\mathrm{FO}^2(\sim, <, +1)$, si il est décidable aura une complexité au moins ExpSpace et probablement plus importante.

Deuxième partie

Patterns

Dans cette partie, on s'intéresse à l'étude d'un formalisme basé sur des combinaisons booléennes de "patterns ". Les patterns sont des objets simples qui permettent d'exprimer des propriétés sur des arbres ou des mots de données. Un pattern peut à la fois exprimer des contraintes sur la structure finie du modèle mais aussi des contraintes d'(in)égalité sur les données. Dans le cas des arbres[12], un pattern d'arbre est constitué d'un arbre étiqueté dont les arêtes sont marquées par la relation enfant $|$ ou descendant $\|$ et de contraintes d'(in)égalité de données entre des nœuds de cet arbre. Un arbre de données satisfait un pattern s'il existe une injection des nœuds du pattern d'arbre dans les nœuds de l'arbre de données qui respecte les contraintes d'étiquettes, de structure ainsi que les contraintes de données. On s'intéresse à deux problèmes en particulier : le model-checking et la satisfaisabilité en présence de contrainte régulière. Une partie des résultats de cette partie est présentée dans [Dav08].

Organisation de la Partie :

Le Chapitre 4 est consacré au cas des arbres. La Section 4.1 contient toutes les définitions et notations nécessaires ainsi que la formulation des deux problèmes étudiés. La section 4.2 est consacrée au problème du model-checking. On montre que dans le cas général le model-checking est dans la classe[13] $\mathrm{P}^{\|\mathrm{NP}}$ et dur pour tout niveau de la hierarchie booléenne BH. On montre aussi que ce problème est déjà NP-complet si on considère un seul pattern d'arbre. La section 4.3 est consacrée au problème de satisfaisabilité en présence de contrainte régulière. On montre que ce problème est indécidable dans le cas général des combinaisons booléennes de patterns d'arbre. On montre qu'on peut obtenir la décidabilité du problème en restreignant les formules ou le pouvoir expressif des patterns d'arbre. On obtient aussi des résultats de décidabilité en restreignant les modèles aux arbres de données de profondeur bornée. Dans tous ces cas, on donne la complexité exacte du problème.

Le Chapitre 5 reprend l'étude du Chapitre 4 dans le cas restreint des mots de données. La première section présente les patterns de mots (restriction des patterns d'arbre à des arbres d'arité un). Les deux sections suivantes étudient respectivement les problèmes de model-checking et de satisfaisabilité en présence de contrainte régulière. Les résultats obtenus sont très similaires à ceux concernant le cas des arbres. Cependant du fait de la structure linéaire des mots la complexité est différente.

Le Chapitre 6 contient un certain nombre de remarques et questions relatives à ce formalisme. La Section 6.1 est consacrée à la définition de possibles extensions des patterns d'arbre. La Section 6.2 contient une discussion sur le problème d'inclusion dont les résultats sont fortement reliés au problème de satisfaisabilité. La Section 6.3 est consacrée à la comparaison de ce formalisme, en terme d'expressivité et de complexité, d'une part avec les formalismes introduits dans la première partie de la thèse et d'autre part avec le langage XPath qui est au cœur de nombreux langages de requête, de transformation et de spécification pour les documents XML. Enfin la Section 6.4 présente des questions ouvertes et leurs enjeux et conclut cette partie.

[12]Les mots représentent la restriction au cas des arbres d'arité 1. On peut ainsi considérer de la même manière des patterns de mots.

[13]La classe $\mathrm{P}^{\|\mathrm{NP}}$ est la classe des langages reconnaissables en temps polynomial par une machine de Turing qui peut poser une série de requêtes en parallèle à un oracle NP.

Chapitre 4

Le cas général des arbres de données

4.1 Définitions

Définition 4.1 Un **pattern d'arbre** $P = (p, \mathcal{C}_\sim, \mathcal{C}_\nsim)$ est formé de :
- un arbre p non ordonné d'arité non bornée dont :
  - les nœuds sont étiquetés soit par une lettre de l'alphabet fini Σ, soit par le symbole wildcard $\star$;
  - les arêtes sont étiquetées soit par $|$ (relation fils) soit par $\|$ (relation descendant),
- deux relations binaires $\mathcal{C}_\sim$ et $\mathcal{C}_\nsim$ sur l'ensemble des nœuds de p.

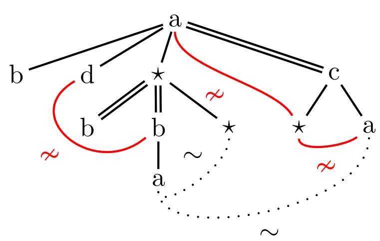
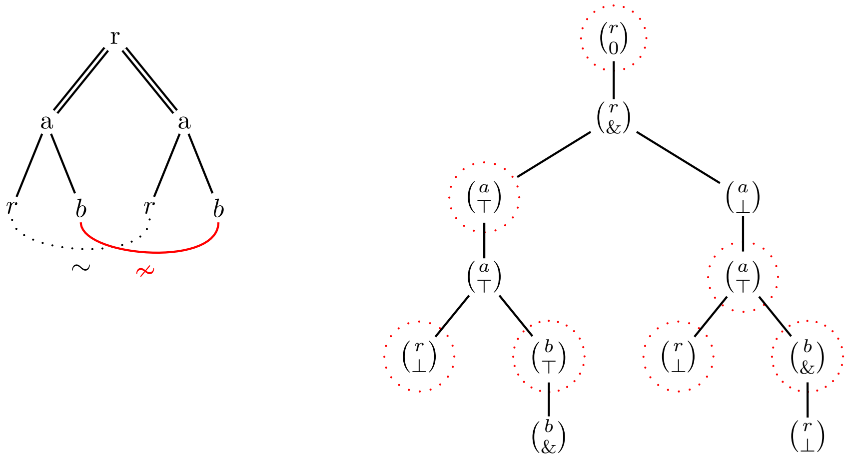


Fig. 4.1 – Exemple de pattern d'arbre

La Figure 4.1 donne un exemple de pattern d'arbre. Les arêtes $|$ correspondent aux arêtes simples et les arêtes $\|$ sont représentées par des arêtes doubles. Les contraintes d'égalité de données sont représentées par des arcs pointillés étiquetés par le symbole $\sim$ qui relient les paires de nœuds qui sont dans la relation $C_\sim$. De la même manière, les contraintes d'inégalité de données sont représentées par des arcs pleins étiquetés par le symbole $\nsim$ qui relient les paires de nœuds qui sont dans la relation $C_\nsim$.

Définition 4.2 Sémantique.
Un arbre de données t satisfait un pattern d'arbre $P = (p, \mathcal{C}_\sim, \mathcal{C}_\nsim)$ s'il existe une fonction injective f des nœuds de p vers les nœuds de t qui respecte les contraintes d'étiquettes, les contraintes de structure (relations descendant/enfant



(a) Exemple de pattern d'arbre (b) Un arbre de données où les positions entourées constituent un témoin du pattern d'arbre

FIG. 4.2 – Exemples

et structure branchante) ainsi que les contraintes d'(in)égalité de données entre les nœuds. On le note $t \models P$. Plus formellement, la fonction f doit vérifier les contraintes suivantes[1] :

- pour tout nœud v de p tel que $v.l \in \Sigma$, on a $v.l = f(v).l$
- pour toute paire (u,v) de nœuds de p, si $(u,v) \in \mathcal{C}_{\sim}$ (resp. $(u,v) \in \mathcal{C}_{\nsim}$) alors $f(u) \sim f(v)$ (resp. $f(u) \nsim f(v)$),
- pour toute paire (u,v) de nœuds de p, si (u,v) est une arête de p étiquetée par $|$ (resp. par $\|$), alors $f(v)$ est un fils (resp. un descendant) de $f(u)$,
- pour tous nœuds u,v,z de l'arbre p, si (u,v) et (u,z) sont deux arêtes de p, alors $f(v)$ et $f(z)$ *ne sont pas* descendants l'un de l'autre dans l'arbre de données t.[2]

Une telle fonction est appelée **témoin** du pattern d'arbre P dans l'arbre de données t. La Figure 4.2 donne un exemple de pattern d'arbre et d'un témoin de ce pattern d'arbre dans un arbre de données.

Dans la suite, on considère aussi des **négations de patterns d'arbre**. Un arbre de données t safisfait la négation d'un pattern P si et seulement s'il n'existe pas de témoin du pattern P dans t.

La propriété d'*injectivité* de la sémantique a des conséquences en terme d'expressivité. Les combinaisons booléennes de patterns d'arbre avec une sémantique injective sont plus expressives que celles obtenues avec une sémantique non injective. Par exemple, avec une sémantique non injective, on ne peut pas assurer qu'un nœud a deux enfants étiquetés de la même manière et appartenant à la

[1] Le symbole wildcard $\star$ correspond à l'absence de contrainte au niveau de l'étiquette du nœud correspondant.

[2] Cette dernière condition assure l'injectivité et le respect de la structure branchante du pattern d'arbre.

même classe. On peut aussi remarquer qu'une conjonction de patterns d'arbre non injectifs est équivalente au pattern d'arbre obtenu en regroupant ces patterns d'arbre au niveau de la racine. En outre tout pattern d'arbre non injectif ou requête conjonctive comme par exemple celles considérées dans [BMS08] peuvent être représentés par une conjonction de disjonctions de patterns d'arbre injectifs modulo une éventuelle explosion exponentielle de la taille de la formule.

Il est à noter que la sémantique ne préserve pas le *plus petit ancêtre commun.* Cette propriété impose que si un nœud du pattern a plusieurs fils, le témoins dans l'arbre de ce nœud est le plus petit ancêtre commun des témoins de ces fils. Par exemple, le témoin donné dans la Figure 4.2 ne vérifie pas cette propriété. En effet le témoin de la racine du pattern n'est pas le plus petit ancêtre commun dans l'arbre des témoins de ces deux fils a. Si à la place, on choisit le nœud $\binom{b}{\&}$ pour témoin de la racine du pattern, le témoin du pattern vérifie la propriété. Les résultats qui suivent restent vrais si la sémantique préserve le *plus petit ancêtre commun.*

Comme indiqué au début de la partie, on considère des arbres non ordonnés. Cependant, les patterns d'arbre n'imposent aucune contrainte sur l'ordre horizontal des nœuds, on peut de la même manière considérer des *arbres de données ordonnés.* Les résultats et les preuves de cette partie sont valables dans les deux cas. En revanche, si on veut ajouter des contraintes de structure horizontale entre les nœuds des patterns d'arbre, la situation est différente. Cela change l'expressivité et peut conduire à l'indécidabilité dans certains cas. On reparle de ce point dans la Section 6.1.3.

Notations : On note $Ptn(\sim, \|, |)$ l'ensemble des patterns d'arbre et $\mathrm{BC}(\sim, \|, |)$ l'ensemble des combinaisons booléennes de patterns d'arbre. On appelle "formule de patterns" une combinaison booléenne de patterns. Dans la suite, on considère des patterns d'arbre restreints n'utilisant que des arêtes $|$ ou que les arêtes $\|$; on appelle respectivement $Ptn(\sim, |)$ et $Ptn(\sim, \|)$ les ensembles de patterns correspondants. De la même façon, on note les ensembles de combinaisons booléennes associés $\mathrm{BC}(\sim, |)$ et $\mathrm{BC}(\sim, \|)$. Enfin, on distingue par BC^+ le fragment positif et par BC^- le fragment négatif. Le fragment positif correspond à l'ensemble des formules de patterns ne contenant pas de négation[3]. Le fragment BC^- correspond à l'ensemble des formules de patterns construites à partir de négations de patterns d'arbre n'utilisant que des disjonctions et des conjonctions.

Définition 4.3 Dans les preuves on utilisera le **Parse Tree** (noté $\mathcal{T}(\varphi)$) correspondant à une combinaison booléenne φ de patterns d'arbre. Les feuilles de cet arbre sont étiquetées par des patterns d'arbre ou des négations de patterns d'arbre et les nœuds internes par les symboles de disjonction ou de conjonction.

Un tel arbre est de taille linéaire en la taille de la formule et peut être construit en temps polynomial.

Contraintes régulières : On considère dans la suite des contraintes régulières sur les arbres de données. Puisque l'on travaille sur des arbres non ordonnés, on considère une version non ordonnée de ce qu'on appelle les "hedge automata" [CDG+07]. Plus précisément, on considère un automate $\mathcal{A}$ bottom-up

[3]Les patterns d'arbre de ces formules peuvent contenir des contraintes d'(in)egalité de données. La négation est interdite uniquement dans la combinaison booléenne.

tel que la transition vers l'état q' dans un nœud étiqueté a est donnée par une combinaison booléenne de clauses de la forme $\#q \leq k$ où q est un état de $\mathcal{A}$ et k une constante entière (encodée en unaire). La clause $\#q \leq k$ est satisfaite si le nœud courant a moins de k fils dans l'état q.

Dans le cas du problème de model-checking, défini ci-dessous, ajouter une contrainte régulière ne change rien à la complexité des résultats. En effet, tester si la projection sur l'alphabet fini d'un arbre de données satisfait la contrainte régulière est un problème PTime et tous les résultats obtenus sont d'une complexité au moins PTime. On ne considèrera donc pas de contraintes régulières pour le problème de model-checking.

Dans la suite on étudie principalement les deux problèmes suivants :

Problème 4.4 Model-Checking : Étant donnés un arbre de données t et une formule de patterns φ, le problème de model-checking consiste à savoir si l'arbre de données t satisfait φ.

Problème 4.5 Satisfaisabilité : Étant données une formule de patterns φ et une contrainte régulière L, le problème de satisfaisabilité consiste à savoir s'il existe un arbre de données qui satisfait φ et dont la projection sur l'alphabet fini appartient au langage L.

4.2 Model-Checking

Dans cette section, on s'intéresse au problème de model-checking, c'est-à-dire, étant donnés un arbre de données et une formule de patterns, on veut savoir si l'arbre satisfait la formule. Le Théorème 4.8 donne la complexité exacte du model-checking d'une combinaison booléenne quelconque de patterns d'arbre. On étudie ensuite la complexité de fragments syntaxiques. Les résultats de cette partie sont basés sur le cas du model-checking d'un seul pattern d'arbre qui est déjà NP-complet. Ce résultat correspond à la Proposition 4.6.

Proposition 4.6 Le model checking d'un pattern d'arbre de $Ptn(\sim, |)$ ou $Ptn(\sim, \|)$ est NP-complet.

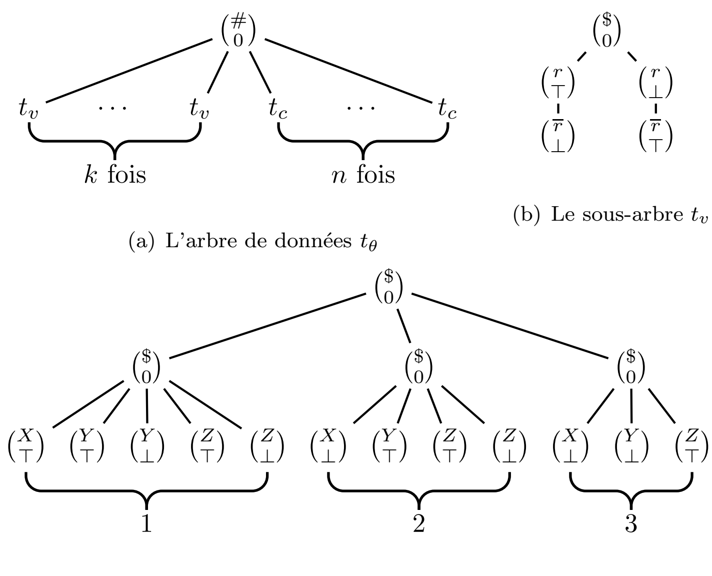
Preuve.
La borne supérieure s'obtient facilement avec un algorithme de type "guess and check". Étant donnés un arbre de données et un pattern d'arbre, on devine un témoin du pattern d'arbre dans l'arbre et on vérifie que ce témoin est correct. Le témoin est de la taille du pattern d'arbre et la vérification se fait en temps polynomial.

La borne inférieure est plus compliquée. On l'obtient par une réduction du problème 3SAT à notre problème. On décrit ici la preuve pour le cas $Ptn(\sim, |)$; la preuve du cas $Ptn(\sim, \|)$ fonctionne exactement de la même manière en remplaçant les étiquettes $|$ des arêtes du pattern d'arbre par des étiquettes $\|$.
Étant donnée une formule propositionnelle θ en forme 3-CNF, on construit, en temps polynomial dans la taille de la formule, un arbre de données t_θ et un pattern d'arbre P_θ de $Ptn(\sim, |)$ tels que $t_\theta \models P_\theta$ si et seulement si θ est satisfaisable.

L'arbre de données étant fixé à partir de la formule, il doit contenir au moins toutes les valuations possibles des variables ainsi que toutes les solutions possibles pour satisfaire chaque clause. Le pattern d'arbre sélectionne une valuation par variable et par clause (ceci est garanti par sa structure) ; les contraintes de données garantissent que la valuation des clauses correspond bien à celle des variables et que cette valuation rend la formule θ vraie.

L'arbre de données t_θ ainsi que la structure d'arbre du pattern d'arbre P_θ dépendent uniquement du nombre de variables et de clauses dans la formule. Ce sont les contraintes de données qui sont spécifiques à la formule θ ; elles représentent les liens entre les variables et les littéraux de chaque clause.

On considère un alphabet fini $\Sigma = \{r, \overline{r}, X, Y, Z, \#, \$\}$. On suppose que la formule propositionnelle θ a k variables et n clauses. L'arbre de données t_θ est composé de k copies de l'arbre t_v, et n copies de l'arbre t_c. Les arbres t_v et t_θ sont décrits dans la Figure 4.3. Chaque copie de t_v correspond à une variable de la formule et chaque copie de t_c correspond à une clause. L'arbre t_θ est composé d'exactement trois classes que l'on nommera par la suite[4] $0, \top, \bot$.



(a) L'arbre de données t_θ

(b) Le sous-arbre t_v

(c) Le sous-arbre t_c

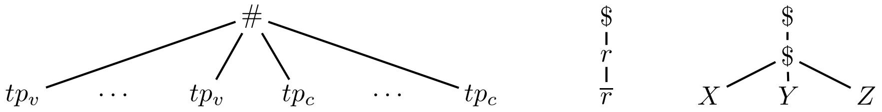
FIG. 4.3 – L'arbre de données t_θ

Chaque sous-arbre t_v, représenté dans la Figure 4.3, contient les deux valuations possibles d'une variable de θ ; la branche gauche représente le cas où la variable est vraie et la branche droite celui où la variable est fausse.

Une clause est la disjonction de trois littéraux que l'on représente ici par X, Y et Z. Chaque sous-arbre t_c (voir Figure 4.3) est formé de trois sous-arbres, chacun d'entre eux représente une des trois possibilités de rendre la clause vraie : (1) soit le premier littéral X est vrai, (2) soit le premier littéral X est faux et le second Y est vrai, (3) soit X et Y sont faux et le troisième littéral Z est vrai.

[4]Ce sont des noms arbitraires choisis pour rendre la preuve plus lisible. En réalité les classes correspondent à trois valeurs de données quelconques mais différentes provenant du domaine infini $\mathcal{D}$.

Le pattern d'arbre $P_\theta = (tp_\theta, \mathcal{C}_\sim, C_\nsim)$ est décrit dans la Figure 4.4. Comme l'arbre de données t_θ, l'arbre tp_θ est constitué de k copies de l'arbre tp_v (chaque copie correspond implicitement à une variable) et n copies de l'arbre tp_c (chacune correspondant implicitement à une clause).



(a) L'arbre tp_θ (b) Les sous-arbres tp_v et tp_c

FIG. 4.4 – L'arbre tp_θ

La structure de l'arbre de données t_θ ainsi que celle de l'arbre tp_θ assurent que tout témoin de P_θ dans t_θ sélectionne exactement une valuation par variable et, pour chaque clause, une valuation correspondant à un cas où la clause est vraie. Il est important de remarquer que cela est dû à la propriété d'injectivité de la sémantique choisie pour les patterns d'arbre.

Il reste maintenant à définir les contraintes de données $\mathcal{C}_\sim$ et $\mathcal{C}_\nsim$ qui vont garantir que la valuation de chaque clause correspond bien à la valuation des variables. Supposons que le premier littéral de la clause c est la variable x (resp. la négation de la variable x). Alors, on ajoute à la relation $C_\sim$ la paire formée du nœud étiqueté r (resp. du nœud $\bar{r}$) du sous-arbre p_v correspondant à la variable x et du nœud étiqueté X du sous-arbre p_c correspondant à la clause c. On fait la même chose avec les autres littéraux de la clause correspondant aux nœuds Y et Z. La Figure 4.5 montre le pattern d'arbre correspondant à la formule θ constituée de l'unique clause $a \vee \neg b \vee c$. La Figure 4.6 donne un autre exemple (dans cette figure toutes les contraintes de données sont du type $\sim$).

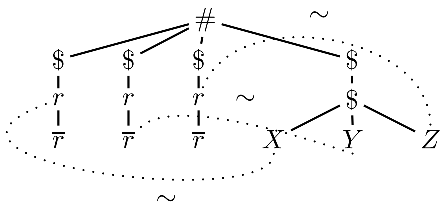


FIG. 4.5 – Le pattern d'arbre correspondant à $\theta = a \vee \neg b \vee c$

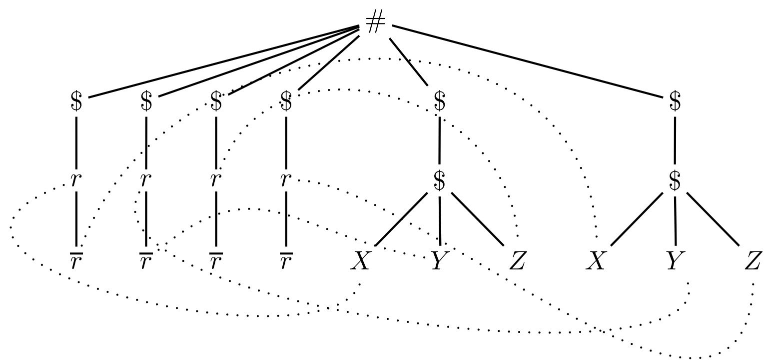


FIG. 4.6 – Le pattern d'arbre pour $\theta = (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d)$.

On prouve maintenant que $t_\theta \vDash P_\theta$ si et seulement si θ est satisfaisable. Supposons que la formule θ est satisfaisable. À partir de n'importe quelle valuation des variables qui rend θ vraie, on construit un témoin de P_θ dans t_θ : le sous-arbre tp_v correspondant à la variable v est associé à la branche gauche du sous-arbre t_v correspondant si la valuation de v est vraie et au sous-arbre droit sinon. Comme chaque clause est satisfaite, on est dans l'un des trois cas décrits dans le sous-arbre t_c et on peut donc faire correspondre les nœuds du sous-arbre tp_c associé à la clause aux nœuds sous-arbre t_c correspondant.
De la même façon, on peut déduire de tout témoin une valuation des variables qui rend la formule vraie. Pour chaque variable x, on considère les nœuds du témoin qui correspondent à la branche de P_θ correspondant à la variable x. Si le nœud r est dans la classe[5] $\perp$, la valuation de x est *faux*, sinon la valuation de la variable x est *vrai*. Vu la structure de l'arbre t_θ et celle du pattern P_θ, cette valuation est une valuation qui rend la formule vraie.

La propriété d'injectivité de la sémantique choisie pour les patterns d'arbre est fortement utilisée dans ce codage. Cependant cette propriété n'est pas indispensable au résultat de complexité. Si on avait choisi une sémantique non injective, on aurait pu faire un codage très similaire en jouant sur les valeurs de données des racines des sous-arbres t_v et t_c. Cela complique légèrement le codage. □

Corollaire 4.7 Le model checking de la négation d'un pattern d'arbre de $Ptn(\sim, |)$ ou $Ptn(\sim, \|)$ est coNP-complet.

On s'intéresse maintenant la complexité du problème de model-checking dans le cas général. On commence par enoncer les résultats (Théorème 4.8). On détaille ensuite les classes de complexités utilisées puis on prouve les différents résultats énoncés.

Théorème 4.8 *Le model-checking de $BC(\sim, |, \|)$ est dans $P^{\|\mathrm{NP}}$. De plus ce problème est difficile pour toute classe BH_k de la hiérachie booléenne* BH.

[5]La classe $\perp$ peut être distinguée de la classe $\top$ car la classe $\top$ est la classe qui apparaît dans une seule position X d'un sous-arbre t_c.

Définition 4.9 La classe $P^{\|NP}$ est la classe des langages reconnaissables en temps polynomial par une machine de Turing qui peut poser une série de requêtes en parallèle à un oracle NP.

Ce type de requêtes sont aussi dénommées requêtes non-adaptatives [wag88]. L'idée est que les requêtes sont indépendantes les une des autres; la liste de l'ensemble des requêtes est formée avant que n'importe laquelle d'entre elle ne soit effectuée.

La hiérarchie booléenne BH est la clôture de la classe NP par les opérations booléennes (union, intersection et complémentaire). On peut alternativement définir cette classe à l'aide de différences de langages [CH86, CGH$^+$88].

Définition 4.10 La hiérarchie booléenne BH est définie récursivement de la manière suivante :
- $BH_1 = NP$;
- pour tout entier $k > 1$, $BH_k = \{L_1 \setminus L_2 \mid L_1 \in NP \text{ et } L_2 \in BH_k\}$;
- $BH = \cup_k BH_k$.

La classe BH est contenue dans la classe $P^{\|NP}$ qui est elle même contenue dans la classe Δ_2 (deuxième niveau de la hiérachie polynomiale) aussi connue comme la classe[6] P^{NP}. La question de savoir si ces inclusions sont strictes est une question difficile ouverte depuis de nombreuses années. On sait cependant que si la hiérachie booléenne s'effondre à n'importe quel niveau alors la hiérarchie s'effondre au niveau trois [CK96].

Proposition 4.11 Le model-checking de $BC(\sim, |, \|)$ est dans $P^{\|NP}$.

Preuve. L'algorithme suivant permet de résoudre le problème de model checking dans $P^{\|NP}$.

Etant donnés un arbre de données t et une formule de patterns φ de $BC(\sim, \|, |)$.
- (1) Evaluer chacun des patterns présents dans la formule φ sur l'arbre t ;
- (2) construire et évaluer le Parse Tree de la formule φ en tenant compte des résultats de (1).

D'après la Proposition 4.6, le model-checking d'un pattern d'arbre est un problème NP. L'étape (1) peut donc être faite en utilisant en parrallèle un oracle NP. Construire le Parse Tree se fait en temps polynomiale et cet arbre est de taille linéaire dans le taille de la formule. Connaissant la valuation des feuilles (étiquetée par un pattern d'arbre ou la négation d'un pattern d'arbre), évaluer le Parse Tree de la formule φ se fait en temps polynomial. Le problème de model-checking est donc bien $P^{\|NP}$. □

Proposition 4.12 Le model-checking de $BC(\sim, |, \|)$ est dur pour toute classe BH_k de la hierarchie booléenne BH.

Preuve. On procède par induction sur $k \in \mathbb{N}$. Le fait que le model checking est $BH_1 = NP$ dur est donné dans la Proposition 4.6. Supposons que le problème de model checking de $BC(\sim, |, \|)$ est BH_k difficile, on montre qu'il est BH_{k+1} difficile.

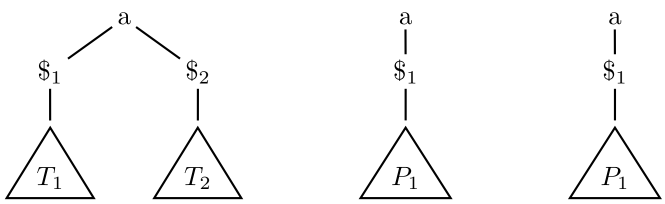
[6] La classe des langages reconnaissables en temps polynomial par une machine de Turing ayant accès sans restriction à un oracle NP.

Soit L un langage BH_{k+1}. Il existe $L_1 \in \mathrm{NP}$ et $L_2 \in \mathrm{BH}_k$ tels que $L = L_1 \setminus L_2$. On considère $w \in L$. Par hypothèse, on peut construire en temps polynomial deux arbres de données T_1 et T_2 et deux formules φ_1 et φ_2 de $\mathrm{BC}(\sim, |, \|)$ tels que :

$$w \in L_1 \text{ ssi } T_1 \models \varphi_1 \text{ et } w \in L_2 \text{ ssi } T_2 \models \varphi_2$$

On construit un arbre de données T et une formule φ tels que $w \in L$ si et seulement si $T \models \varphi$.

Pour simplifier la construction, on ajoute à l'alphabet Σ deux nouvelles lettres $\$_1$ et $\$_2$. On peut effectuer une réduction similaire sans augmenter la taille de l'alphabet fini[7].
La racine de T (voir Figure 4.7) est étiquetée par une lettre a arbitrairement choisie dans Σ et possède deux enfants étiquetés respectivement $\$_1$ et $\$_2$. Les classes de ces nœuds n'ont pas d'importance. On place l'arbre T_1 sous le nœud $\$_1$ et l'arbre T_2 sous le nœud $\$_2$.



(a) L'arbre T (b) Le pattern P_1 (c) Le pattern P_2

FIG. 4.7 –

Étant donné un pattern P de la formule φ_1 on construit un nouveau pattern P_1 en ajoutant deux nœuds au pattern P (voir Figure 4.7). La racine de P_1 est un nouveau nœud étiqueté a et possède un seul fils étiqueté $\$_1$ qui aussi un nouveau nœud. Le fils du nœud $\$_1$ est la racine du pattern P. Les deux nouvelles arêtes ainsi rajoutées sont étiquetées $|$ et les contraintes de données sont inchangées. On appelle φ_1' la formule φ_1 dans laquelle on a remplacé chaque pattern P par le pattern P_1 précédemment construit. Grâce au nœud $\$_1$ dans T et dans les patterns de φ_1', on a :

$$T \models \varphi_1' \text{ ssi } T_1 \models \varphi_1$$

De la même manière on peut construire une formule φ_2' à partir de φ_2 en ajoutant à chacun des patterns correspondant un nœud a et un nœud $\$_2$ afin d'avoir :

$$T \models \varphi_2' \text{ ssi } T_2 \models \varphi_2$$

On définit la formule φ comme la formule $\varphi_1' \wedge \neg\varphi_2'$. D'après ce qui précède, on a :

$$T \models \varphi \text{ ssi } T_1 \models \varphi_1 \text{ et } T_2 \models \varphi_2$$

[7]Un codage plus lourd peut être donné en remplaçant respectivement les nœuds $\$_1$ et $\$_2$ de T par des branches de nœuds d'une même classe dans T et en ajoutant dans les patterns de φ les contraintes de données correspondantes.

c'est-à-dire

$$T \models \varphi \text{ ssi } w \in L_1 \backslash L_2 = L.$$

La construction est PTime. Le problème de model checking de $\mathrm{BC}(\sim, |, \|)$ est donc BH_{k+1} difficile. □

On peut remarquer que dans la preuve de la Proposition 4.6, le pattern d'arbre utilisé dans la borne inférieure utilise uniquement des arêtes $|$. On pourrait faire la même preuve avec le même codage en remplaçant les arêtes $|$ par des arêtes $\|$ dans le pattern d'arbre. On en déduit le résultat suivant :

Proposition 4.13 Le model-checking pour les fragments $\mathrm{BC}(\sim, \|)$ et $\mathrm{BC}(\sim, |)$ est complet pour toute classe de la hiérarchie booléenne BH.

Proposition 4.14 Le model-checking pour les fragments $\mathrm{BC}^+(\sim, \|)$, $\mathrm{BC}^+(\sim, |)$ et $\mathrm{BC}^+(\sim, |, \|)$ est NP-complet.

Proposition 4.15 Le model-checking pour les fragments $\mathrm{BC}^-(\sim, \|)$, $\mathrm{BC}^-(\sim, |)$ et $\mathrm{BC}^-(\sim, |, \|)$ est coNP-complet.

Remarque 4.16 *O*n obtient les mêmes résultats si on enlève la contrainte d'*injectivité* de la sémantique des patterns d'arbre. La preuve de la borne inférieure de la Proposition 4.6 est très légèrement différente. Toutes les autres preuves s'obtiennent de la même manière.

La preuve de NP-dureté du model-checking d'un pattern d'arbre dans le cas non injectif est toujours une réduction du problème 3SAT. L'idée est la suivante : on considère le codage utilisé pour prouver que le model-checking d'un seul pattern d'arbre est NP-difficile. Si on oublie la contrainte d'injectivité, rien ne nous assure que le pattern d'arbre P_θ sélectionne bien une valuation pour chaque variable et chaque clause. Pour corriger ce problème, il suffit de modifier légèrement le codage afin que chaque racine d'un sous-arbre t_v ou t_c appartienne à une classe différente au lieu d'appartenir toutes à la même classe (notée 0 dans la preuve). Il suffit alors de rajouter les contraintes de données correspondantes[8] dans le pattern d'arbre P_θ et la preuve se déroule de la même manière que dans le cas injectif. Intuitivement, mettre les racines des sous-arbres dans des classes différentes permet de distinguer les sous-arbres les uns des autres et de forcer ainsi l'injectivité dans le pattern d'arbre.

4.3 Satisfaisabilité

Dans cette partie, on s'intéresse au problème de satisfaisabilité : "Étant donnés une formule de patterns et un langage régulier, existe-t-il un arbre de données qui vérifie la formule de patterns et dont la projection sur l'alphabet fini appartienne au langage régulier ?". On commence par montrer que ce problème est indécidable dans le cas général. On détaille ensuite les résultats de décidabilité et de complexité pour différentes restrictions du problème.

[8]On remarque que cela introduit des contraintes d'inégalité de données dans le pattern P_θ qui contenait uniquement des contraintes d'égalité de données dans la version avec contrainte d'injectivité.

4.3.1 Indécidabilité

Théorème 4.17 *Le problème de satisfaisabilité pour BC*($\sim, |, \|$) *en présence de contrainte régulière est indécidable.*

Preuve. On réduit le problème de l'arrêt des machines à deux compteurs (aussi appelées machines de Minsky) au problème de satisfaisabilité. On rappelle que les machines de Minsky sont des machines à deux compteurs pouvant effectuer des opérations d'incrément et de décrément ainsi que des tests à zéro sur les compteurs. Le problème de l'arrêt de ses machines est indécidable [Min67].

Étant donnée une machine $\mathcal{M}$, on construit un langage régulier $L_{\mathcal{M}}$ et une formule de patterns $\varphi_{\mathcal{M}}$ tels que les modèles de la formule de patterns dont la projection sur l'alphabet fini appartient au langage régulier correspondent aux codages des exécutions acceptantes de la machine.

Dans la suite, on décrit le codage d'une exécution de la machine $\mathcal{M}$ en un arbre de données puis on décrit le langage $L_{\mathcal{M}}$ et la formule de patterns $\varphi_{\mathcal{M}}$.

Le codage d'une exécution peut être découpé en trois parties :

1. la structure générale de l'arbre qui va être imposée par le langage régulier ;
2. la structure de données propre à chaque configuration ;
3. l'évolution de la structure de données entre deux configurations successives.

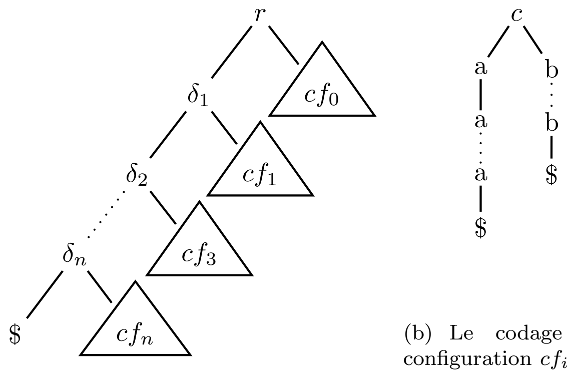
La structure générale de l'arbre est décrite dans la Figure 4.8(a). Le codage d'une exécution est constitué d'une branche étiquetée par la suite des transitions. Chaque nœud-transition contient un sous-arbre qui correspond à la valeur des compteurs dans la configuration correspondante. Ce sous-arbre (voir Figure 4.8(b)) est constitué de deux branches, une branche (dite branche a) dont les nœuds internes sont étiquetés a et la feuille finale est étiquetée par la lettre \$, et une branche (dite branche b) dont les nœuds internes sont étiquetés b et la feuille est étiquetée \$. Ces deux branches représentent la valeur en unaire de chacun des compteurs. La structure générale de l'arbre est contrôlée par le langage régulier $L_{\mathcal{M}}$ qui assure que l'arbre est bien une suite de configurations codées de la manière décrite précédemment. Il permet aussi de vérifier le résultat des tests à zéro et assure que la suite des transitions respecte les règles de la machine $\mathcal{M}$ (suite d'états et configurations initiale et finale).

Les données permettent de contrôler l'évolution de la valeur des compteurs entre deux configurations successives. Pour cela il faut avoir une certaine structure au niveau des valeurs de données et la continuité de cette structure tout au long d'une exécution. On contraint cette structure à l'aide de patterns d'arbre. On explique comment le faire sur une branche a, les branches b respectent la même structure qui peut être assurée à l'aide de patterns d'arbre du même type.

Toutes les positions de la branche a d'une configuration sont dans des classes différentes. Cette propriété est assurée en interdisant le pattern d'arbre de la Figure 4.9(a) qui correspond au cas où deux a d'une même configuration ont la même valeur de donnée.

Entre deux configurations successives, on place les données de telle sorte que si deux nœuds a sont dans des positions identiques[9] dans deux configurations

[9]On dit que deux nœuds a sont dans des positions identiques s'ils correspondent chacun au k-ième nœud a d'un sous-arbre codant une configuration pour un même entier k.



(b) Le codage d'une configuration cf_i

(a) La codage d'une exécution

FIG. 4.8 – Codage d'une exécution

successives, alors ils sont dans la même classe. Cette propriété est assurée de manière inductive en interdisant les deux patterns d'arbre de la Figure 4.9(b).

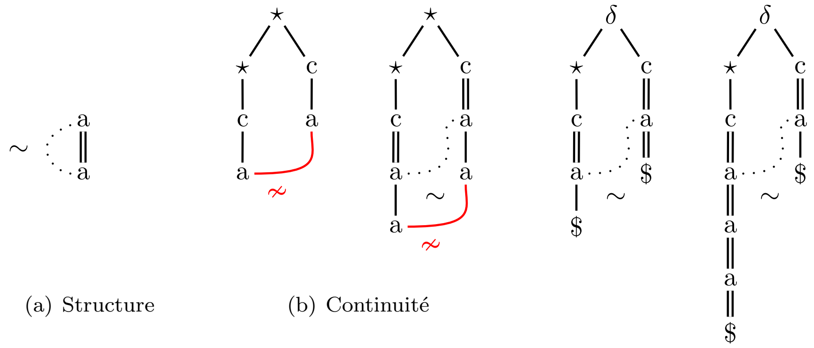
La dernière étape consiste à contrôler la longueur des branches c'est-à-dire la valeur des compteurs le long de l'exécution. Celles-ci doivent être cohérentes avec les transitions de la machine. Entre deux configurations successives, l'incrémentation d'un compteur va correspondre à la présence d'exactement un nœud a (resp. b) supplémentaire dans la branche correspondante. On assure ce comportement pour chaque couple de configurations successives en interdisant tout comportement violant la transition correspondante. Cela ne dépend pas de la valeur du compteur de la configuration courante mais de l'opération effectuée sur le compteur (à savoir si celui-ci sera incrémenté, décrémenté ou inchangé dans la configuration suivante). On peut donc représenter les comportements indésirables à l'aide d'un nombre fini[10] de patterns d'arbre que l'on interdit dans la formule. La Figure 4.9(c) décrit le cas d'une transition qui incrémente le premier compteur. Le premier pattern impose que la valeur de donnée du dernier a de la nouvelle configuration soit différente de toutes celles des a de l'ancienne configuration ; cela assure que le compteur a été incrémenté. Le deuxième pattern assure que le compteur n'a pas été incrémenté de plus de 1. Les patterns correspondants aux autres cas se construisent de la même façon.

Nous avons maintenant tous les éléments de la réduction. La formule $\varphi_{\mathcal{M}}$ est la conjonction de la négation des patterns d'arbre décrits précédemment. Par construction les modèles de la formule $\varphi_{\mathcal{M}}$ qui respectent le langage régulier $L_{\mathcal{M}}$ sont exactement les codages des exécutions acceptantes de la machine $\mathcal{M}$.

Le problème de satisfaisabilité des combinaisons booléennes de patterns d'arbre est donc indécidable. $\square$

Remarque 4.18 Ce résultat est aussi valable si on n'impose pas la contrainte d'*injectivité*. L'idée est la même que dans le cas du codage de 3SAT utilisé dans

[10]dépendant du nombre de transitions de la machine



(a) Structure

(b) Continuité

(c) Incrémentation du premier compteur

FIG. 4.9 – Les patterns d'arbre interdits

les preuves de model-checking. On doit imposer que les positions c soient toutes dans des classes différentes afin de distinguer les différentes branches de l'arbre. Un problème supplémentaire apparaît car la relation $\|$ autorise, dans ce cas, les deux nœuds mis en relation à correspondre au même nœud dans l'arbre de données. Il faut donc aussi modifier légèrement les patterns d'arbre qui utilisent cette relation. Par exemple pour exprimer le fait que tous les a d'une branche sont dans des classes différentes, il faut deux patterns d'arbre ; l'un d'entre eux interdit que deux a consécutifs soient dans la même classe ; l'autre interdit que les descendants d'un fils d'un a soient dans la même classe que ce a. On utilise le même genre d'astuce pour modifier les autres patterns d'arbre utilisés.

Remarque 4.19 La formule de patterns utilisée dans la preuve du Théorème 4.17 utilise uniquement une conjonction de négations de patterns d'arbre.

Théorème 4.20 *Le problème de satisfaisabilité pour* $BC^{-}(\sim, |, \|)$ *en présence de contrainte régulière est indécidable.*

On considère maintenant des restrictions naturelles de notre problème afin d'obtenir des résultats de décidabilité. Dans la sous-Section 4.3.2 on s'intéresse aux cas où l'on restreint le pouvoir expressif des patterns d'arbre ou la forme des formules considérées. Dans la sous-Section 4.3.3 on restreint les modèles en considérant uniquement des arbres de données de profondeur bornée.

4.3.2 Restrictions des formules

Dans cette partie, on s'intéresse au problème de satisfaisabilité pour des fragments de $BC(\sim, |, \|)$. Une première façon de restreindre le problème est de restreindre les formules. Le Théorème 4.20 montre que le problème de satisfaisabilité est aussi indécidable pour le fragment négatif. En revanche si on interdit la négation dans les formules, on obtient la décidabilité. La Proposition 4.21 donne la complexité exacte du problème pour le fragment positif. Une autre restriction possible concerne le pouvoir expressif des patterns d'arbre. On obtient aussi dans ce cas des résultats de décidabilité. Les Théorèmes 4.24 et 4.27 donnent les complexités exactes des fragments obtenus en considérant des patterns d'arbre n'utilisant qu'un seul type d'arête (soit $|$ soit $\|$).

La preuve du Théorème 4.17 utilise les négations de patterns d'arbre afin de garantir des propriétés de type universel. Une formule de patterns du fragment positif demande d'assurer l'existence de certains patterns d'arbre dans le modèle, ce qui est intuitivement plus simple.

Fragment positif.

Proposition 4.21 Le problème de satisfaisabilité en présence de contrainte régulière est NP-complet pour les fragments $\mathrm{BC}^+(\sim, |, \|)$, $\mathrm{BC}^+(\sim, |)$ et $\mathrm{BC}^+(\sim, \|)$.

Preuve. Les bornes inférieures sont faciles à obtenir. Voici une preuve possible pour le fragment $\mathrm{BC}^+(\sim, |)$ qui utilise la conjonction de deux patterns d'arbre.

On considère le codage de 3SAT utilisé dans la preuve de la Proposition 4.6. Étant donnée une formule propositionnelle θ, on choisit le langage L constitué de l'unique arbre correspondant à la projection sur l'alphabet fini de l'arbre t_θ codant la formule. On appelle P_{t_θ}, le pattern d'arbre obtenu à partir de t_θ en remplaçant les valeurs de données par les contraintes de données correspondantes et en étiquetant toutes les arêtes par $|$. Par construction un seul arbre vérifie le pattern d'arbre P_{t_θ} tout en respectant L ; il s'agit de l'arbre t_θ. On en déduit que la formule de patterns $P_\theta \wedge P_{t_\theta}$ est satisfaisable en présence du langage L si et seulement si la formule propositionnelle θ est satisfaisable.

Comme dans la preuve de la Proposition 4.6, on obtient le résultat correspondant au fragment $\mathrm{BC}^+(\sim, \|)$ en remplaçant les arêtes $|$ des patterns d'arbre par des arêtes $\|$.

La borne supérieure s'obtient pour $\mathrm{BC}^+(\sim, |, \|)$ de la même façon que dans la preuve du Théorème 4.5 de [BFG05]. Elle est plus complexe que la borne inférieure car il est possible que tous les modèles d'un langage régulier L soient de taille au moins exponentielle en $|L|$. Cependant, l'absence de négation permet de se concentrer uniquement sur les positions témoins des patterns d'arbre. Ainsi, il suffit de deviner une petite partie du modèle correspondant à ces témoins (de taille polynomiale) et de vérifier que cet arbre peut être complété en un modèle valide sans avoir besoin de le construire.

On considère une formule φ de $\mathrm{BC}^+(\sim, |, \|)$ et un langage régulier L. À l'aide d'un argument de pompage vertical et horizontal, on peut montrer que s'il existe un modèle de φ respecte le langage L, il en existe un d'arité et de profondeur polynomiales dans la taille de la formule de patterns et du langage régulier L. On détaille ici ces arguments qui seront réutilisés dans d'autres preuves.

On rappelle que la contrainte régulière L est donnée par un automate $\mathcal{A}$ bottom-up tel que la transition vers l'état q' dans un nœud étiqueté a est donnée par une combinaison booléenne de clauses de la forme $\#q \leq k$ où q est un état de $\mathcal{A}$ et k une constante entière (encodée en unaire). La clause $\#q \leq k$ est satisfaite si le nœud courant a moins de k fils dans l'état q.

On considère un modèle T dont on marque les nœuds d'un témoin pour chaque pattern de la formule φ. On colorie les positions de l'arbre à l'aide d'états de $\mathcal{A}$ correspondant à une exécution acceptante. On appelle $\overline{T}$ l'arbre T ainsi colorié.

Pompage vertical : Supposons qu'il existe dans $\overline{T}$ deux nœuds u et v étiquetés par le même état de $\mathcal{A}$ tels que u et v ne sont pas des nœuds témoins, v est un descendant de u et tout descendant de u qui est un nœud témoin dans l'arbre est aussi un descendant de v. Alors, l'arbre obtenu en remplaçant u et son sous-arbre par v et son sous-arbre est modèle de la formule φ. Le coloriage correspond toujours à une exécution de l'automate $\mathcal{A}$. Les nœuds témoins ainsi que leur ordre respectif sont préservés, donc ces nœuds consituent toujours des témoins pour les patterns de la formule.
Pompage horizontal : On considère un nœud v de $\overline{T}$ la transition de l'exécution de $\mathcal{A}$ entre v et ces fils et l'ensemble des contraintes du type $\#q > k$ utilisées pour valider cette transition. Si v a plus de $k+1$ fils coloriés q, l'arbre obtenu en supprimant les nœuds surnuméraires dont le sous-arbre correspondant ne contient aucun nœud témoin est toujours un modèle de la formule φ. Le coloriage décrit toujours une exécution acceptante de $\mathcal{A}$ et les nœuds témoins sont conservés.

Ces deux mécanismes de pompages garantissent que si une formule φ a un modèle qui respecte une contrainte régulière donnée par un automate $\mathcal{A}$ alors, il existe un modèle de profondeur bornée et d'arité polynomiale dans la taille de φ et de $\mathcal{A}$.

La profondeur peut être bornée par $N.|Q|+N$ où N est la somme du nombre de nœuds des patterns de la formule et Q le nombre d'états de $\mathcal{A}$. L'arité peut être bornée par $N+K$ où K est le maximum sur l'ensemble des transitions de $\mathcal{A}$ de la somme de $k+1$ pour chaque contrainte $\#q \leq k$ (ou $\#q < k$) apparaissant dans la transition.

On revient maintenant à la preuve. On considère donc un modèle de φ qui respecte le langage L d'arité et de profondeur polynomiales dans la taille de la formule de patterns et du langage L. on considère le sous-arbre minimal qui contient toutes les positions témoins, c'est-à-dire le plus petit ensemble de branches de l'arbre qui contient toutes les positions témoins. Cet arbre est de taille polynomiale.

Si on devine un tel sous-arbre minimal t, il faut pouvoir vérifier la propriété suivante : il existe un arbre t' dont la projection sur l'alphabet fini appartient au langage régulier L et tel que en supprimant des sous-arbres dans t', on obtient l'arbre t. Pour faire cela, on devine aux nœuds de t les états et les transitions correspondant à une exécution acceptante de l'automate sur un arbre t'. Il suffit ensuite de vérifier en chaque nœud si la combinaison booléenne correspondant à la transition peut être vérifiée en ajoutant des nœuds à l'arbre t. Comme la formule de patterns φ ne contient pas de négation de pattern d'arbre, l'ajout de nœuds dans l'arbre ne change pas le caractère satisfaisable ou insatisfaisable de la formule.

L'algorithme suivant résout le problème de la satisfaisabilité d'une formule de patterns sans négation en présence de contrainte régulière : (1) deviner l'arbre de taille polynomiale, les positions témoins ainsi que les états et transitions de l'automate, (2) vérifier que les témoins sont corrects et que l'arbre peut être complété en un modèle du langage régulier. Cet algorithme est NP. □

Remarque 4.22 La borne inférieure de la Proposition 4.21 peut déjà être obtenue dans le cas d'arbres sans données. On peut par exemple regarder le codage proposé dans la preuve du Théorème 4.5 de [BFG05].

Proposition 4.23 Le problème de satisfaisabilité de $Ptn(\sim, |, \|)$ en présence de contrainte régulière est PTIME.

Preuve. L'idée est la suivante : Si les contraintes de données présentes dans le pattern d'arbre ne se contredisent pas, on montre que l'on peut se ramener à au problème sans valeur de données.
On considère un pattern d'arbre $P = (p, \mathcal{C}_\sim, C_\nsim)$ et un langage régulier L. On suppose que les relations $C_\sim$ et $C_\nsim$ ne se contredisent pas. Le problème est équivalent dans ce cas au problème sans donnée. En effet s'il existe un arbre du langage régulier qui contient le motif décrit par p alors on peut contruire un arbre de données à partir de ce modèle et d'un témoin de p en plaçant les données en respectant les contraintes $C_\sim$ et $C_\nsim$ dans les nœuds du témoin de p. Les positions qui correspondent au témoin de p dans l'arbre sans donnée constituent un témoin de P dans l'arbre de données. La réciproque est évidente.

Sans données, le problème correspond au vide du produit d'un automate reconnaissant le langage régulier et d'un automate reconnaissant le motif p. Ceci se fait en temps polynomial.

Étant donné un pattern d'arbre, le problème de savoir si ses contraintes de données sont cohérentes se résout en temps polynomial. En effet, il suffit de marquer les ensembles de nœuds qui doivent être dans la même classe d'après la relation $C_\sim$, on vérifie ensuite qu'aucune contrainte de $C_\nsim$ ne contredit $C_\sim$. □

La preuve du Théorème 4.17 utilise de manière significative les arêtes | et ‖ afin de compter la valeur des compteurs (a priori non-bornée). Il n'est donc pas surprenant que la restriction de l'utilisation de ces arêtes mène à la décidabilité. Utiliser un seul type d'arête permet néanmoins, à l'aide d'un pattern d'arbre de taille polynomiale, de comparer les positions d'un arbre de profondeur polynomiale. Cette idée est la clé des bornes inférieures des Théorèmes 4.24 et 4.27. À l'aide de cette idée, on peut coder des configurations de machine de Turing de taille exponentielle dans les feuilles d'un arbre de profondeur polynomiale.

Fragment descendant.

Théorème 4.24 *Le problème de satisfaisabilité en présence de contrainte régulière pour le fragment $BC(\sim, \|)$ est* NEXPTIME*-complet.*

Preuve. La borne supérieure vient d'une propriété de plus petit modèle. Étant donnés une formule de patterns et un langage régulier, s'il existe un modèle de la formule de patterns qui respecte le langage régulier alors il en existe un de d'arité et de profondeur polynomiales dans la taille de l'entrée. Il existe donc un modèle de taille exponentielle.

On considère une formule de patterns φ et un langage régulier L. Soit t un modèle de φ qui respecte le langage L. On marque les nœuds d'un témoin de chaque pattern d'arbre de φ qui est satisfait par t. Comme dans la preuve de la Proposition 4.21, on peut couper l'arbre horizontalement et verticalement en évitant les positions marquées et en respectant l'appartenance au langage régulier L. On obtient ainsi un arbre d'arité et profondeur polynomiales dans la taille de l'entrée qui respecte la contrainte régulière et contient toujours les témoins des patterns d'arbre précédemment marqués. Comme les patterns d'arbre de φ ne contiennent que des arêtes ‖, si l'arbre d'origine ne contenait pas de témoin d'un pattern d'arbre, le nouvel arbre n'en contient pas non plus. Le nouvel arbre est donc toujours un modèle de la formule de patterns φ.

L'algorithme naturel donne donc la réponse au problème de satisfaisabilité en temps exponentiel. On commence par deviner un arbre t de taille au plus exponentielle. Pour chaque pattern d'arbre de la formule, on considère une à une chaque correspondance possible du pattern d'arbre dans l'arbre (il y a $|t|^k$ solutions possibles si k est la taille du pattern d'arbre) et on teste s'il s'agit ou non d'un témoin du pattern d'arbre (en temps polynomial dans la taille de l'arbre). Enfin, on évalue le Parse Tree de la formule de patterns φ (en temps polynomial dans la taille de la formule).

La borne inférieure s'obtient par la réduction du problème d'acceptance des machines de Turing non-déterministes qui s'exécutent en temps exponentiel à notre problème. On code les exécutions de la machine dans des arbres de données. On construit ensuite un langage régulier et une formule de patterns dont les modèles sont exactement les exécutions acceptantes de la machine sur une entrée donnée.

On considère une machine de Turing $\mathcal{M}$ et une entrée w de taille n. La machine $\mathcal{M}$ s'éxecute en temps 2^n sur l'entrée w. Une exécution acceptante de $\mathcal{M}$ sur w est donc une suite d'au plus 2^n transitions $\delta_0, \delta_1, \delta_2, \delta_3, \ldots, \delta_f$. De plus chaque configuration est de taille au plus 2^n. On code une telle exécution dans un arbre de données dont la forme est donnée dans la Figure 4.10.

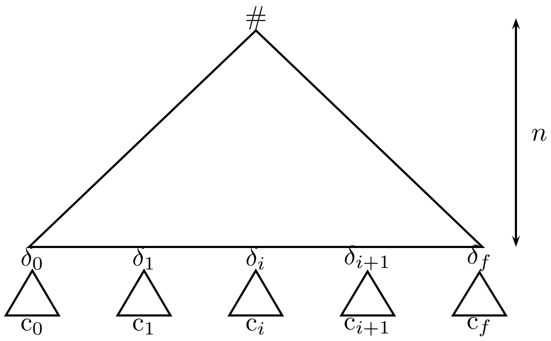


Fig. 4.10 –

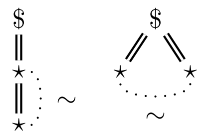
La première partie de l'arbre (de racine #) est un arbre binaire de profondeur n. Chaque nœud interne de l'arbre a exactement deux fils, l'un étiqueté 0 et l'autre étiqueté 1. Chaque feuille de cet arbre, est étiquetée par une transition de la machine et est suivie d'un arbre binaire de profondeur n (de racine \$) tel que le mot des feuilles représente une configuration de la machine (on considère les feuilles dans l'ordre lexicographique des chemins à partir de la racine du sous-arbre). Dans la suite, on appelle ces sous-arbres des $\mathcal{C}$-sous-arbres. Tous les $\mathcal{C}$-sous-arbres ont exactement la même structure au niveau des données ainsi que pour les étiquettes des nœuds internes. La seule différence se fait au niveau des feuilles dont les étiquettes permettent de coder les configurations. Pour pouvoir identifier les configurations, on met dans un même $\mathcal{C}$-sous-arbre une donnée différente à chaque nœud.

On décrit maintenant le langage régulier et la formule de patterns vérifiant la propriété que les modèles de la formule de patterns qui respectent le langage sont exactement le codage des exécutions acceptantes de la machine $\mathcal{M}$ sur l'entrée w.

Le langage régulier assure que l'arbre est un sous-arbre binaire de profondeur n étiqueté par 0, 1 dont les feuilles contiennent chacune à nouveau un sous-arbre binaire de profondeur n étiqueté par 0, 1. La formule de patterns fixe la structure de données. Elle vérifie aussi les contraintes initiales et finales[11] et vérifie que les transitions sont respectées au niveau des configurations. Les feuilles des $\mathcal{C}$-sous-arbres sont identifiées de manière unique par la suite de données correspondant au chemin à partir de la racine (qui est la même pour tous les $\mathcal{C}$-sous-arbres). Cela permet de comparer l'évolution de l'étiquetage de ces positions entre deux configurations successives à l'aide de patterns d'arbre. Comme dans la preuve du Théorème 4.17, on va interdire les patterns d'arbre correspondant aux comportements indésirables.

La première étape est donc de fixer la structure de données.

- Dans un $\mathcal{C}$-sous-arbre, chaque nœud appartient à une classe différente. On assure cette propriété en interdisant les patterns d'arbre donnés dans la Figure 4.11.



(a) (b)

FIG. 4.11 –

- Les nœuds des $\mathcal{C}$-sous-arbres correspondant à une même position (c'est-à-dire au même chemin depuis la racine \$ de leur $\mathcal{C}$-sous-arbre) appartiennent à la même classe. Cette propriété est assurée en interdisant pour tout k les patterns d'arbre donnés dans la Figure 4.12.

La seconde étape consiste à assurer que les transitions de la machine sont simulées correctement dans l'arbre de données. Si on sait parler de trois positions consécutives d'une configuration, il suffit d'interdire les comportements indésirables de ces groupes de positions entre deux configurations successives.

On peut distinguer deux positions consécutives de la manière suivante : deux positions a et b satisfont le pattern d'arbre décrit dans la Figure 4.13(a) si et seulement si b est la position successeur de a dans le $\mathcal{C}$-sous-arbre de racine \$. De la même façon, si les positions a, b et c satisfont le pattern d'arbre décrit dans la Figure 4.13(b), ce sont des positions consécutives dans le $\mathcal{C}$-sous-arbre correspondant.

À l'aide de ces patterns d'arbre, on peut construire les patterns d'arbre correspondant aux comportements indésirables. Un exemple est donné dans la Figure 4.14.

[11]Ces contraintes peuvent aussi être inclues dans la langage régulier. Pour exprimer la contrainte initiale à l'aide de patterns, on a besoin de pattern positif afin d'assurer l'existence de certains nœuds. La contrainte finale peut s'exprimer uniquement à l'aide de contraintes négatives.

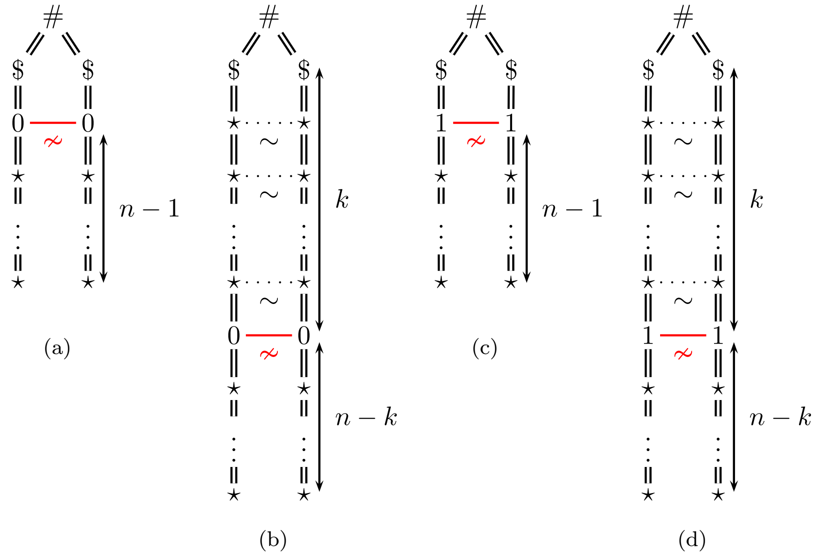


Fig. 4.12 –

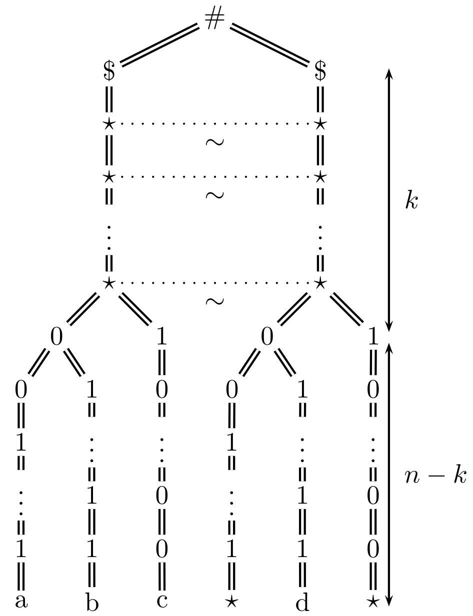
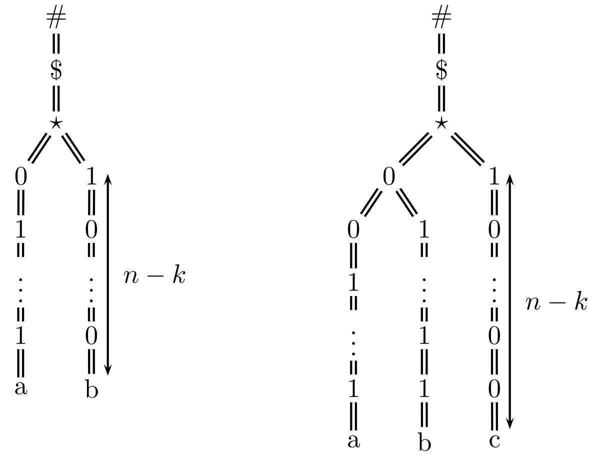


Fig. 4.14 – Exemple de comportement indésirable

En tout, on a besoin d'un nombre polynomial de patterns d'arbre interdits pour assurer d'une part la structure de donnée et d'autre part l'évolution correcte des configurations dans l'arbre.

On peut donc construire un langage régulier et une formule de patterns de taille polynomiale dans la taille de la machine $\mathcal{M}$ et de l'entrée w tels que les modèles de la formule de patterns qui respectent le langage régulier sont



(a) Les positions a et b sont consécutives.

(b) a, b et c sont consécutives.

FIG. 4.13 – Comment distinguer des positions consécutives.

exactement le codage des exécutions acceptantes de la machine $\mathcal{M}$ sur l'entrée w.
□

Remarque 4.25 Comme dans la preuve d'indécidabilité, la preuve de NEXPTIME-dureté utilise une formule du fragment négatif de BC($\sim, |, \|$).

On déduit de cette remarque le résultat suivant :

Proposition 4.26 Le problème de satisfaisabilité en présence de contrainte régulière pour le fragment BC$^-$($\sim, \|$) est NEXPTIME-complet.

Fragment Successeur.

Théorème 4.27 *Le problème de satisfaisabilité de BC($\sim, |$) en présence de contrainte régulières est* 2EXPTIME*-complet.*

Les Lemmes 4.28 et 4.29 donnent respectivement les résultats de borne supérieure et inférieure.

Lemme 4.28 Le problème de satisfaisabilité de BC($\sim, |$) en présence de contrainte régulières est 2EXPTIME.

Preuve. On commence par montrer qu'une formule de patterns de BC($\sim, |$) a un modèle si et seulement si elle a un modèle dont le nombre de classes est au plus doublement exponentiel dans la taille de la formule. On montre ensuite que l'on peut reconnaître la projection sur l'alphabet fini de ces modèles particuliers à l'aide d'un automate fini de taille doublement exponentielle dans la taille de la formule et du langage. La construction de l'automate ne demande pas plus que 2EXPTIME et le vide d'un tel automate étant PTIME. On obtient ainsi la borne supérieure 2EXPTIME.

On considère une formule de patterns φ, un langage régulier L et un arbre de données t qui satisfait la formule et respecte le langage. On divise les patterns d'arbre de φ en deux groupes : on appelle patterns d'arbre positifs les patterns d'arbre qui apparaissent dans l'arbre t et patterns d'arbre négatifs, ceux qui n'apparaissent pas. On marque dans l'arbre les nœuds d'un témoin de chacun des patterns d'arbre positifs.

On commence par remarquer que l'on peut restreindre l'arité du modèle grâce à l'argument de pompage utilisé dans la preuve du Théorème 4.24. En revanche, l'argument de pompage vertical ne peut pas être utilisé dans ce cas. Les patterns d'arbre contenant des relations enfant $\mid$ entre les nœuds, pomper verticalement pourrait introduire des témoins pour les patterns d'arbre négatifs. On peut donc assurer qu'il existe un modèle t_r d'arité bornée par une constante[12] r (polynomiale dans la taille de L et φ).

À partir de ce modèle t_r, on peut construire un modèle t_{rc} qui aura la même projection sur l'alphabet fini mais un nombre de classes borné par une constante exponentielle de la forme $m = p + r^{d+1}$ où d représente la somme des profondeurs des patterns d'arbre négatifs et p le nombre de nœuds des patterns d'arbre positifs.

On considère un témoin pour chaque pattern positif. On peut recolorier les valeurs de données de l'arbre t_r à l'aide de m classes de telle sorte que les relations d'égalité entre les données des nœuds soient préservées pour tous les sous-arbres de profondeur d. Afin de préserver la présence des témoins des patterns d'arbre positifs, on ne modifie pas les positions dont la classe apparaît dans un des témoins. Le recoloriage se fait de la racine vers les feuilles à chaque étape, on colorie les nœuds à profondeur k en fonction du recoloriage des autres nœuds. Plus précisément, pour tout nœud v à profondeur $k - d$, on colorie ses descendants à la profondeur k afin que les relations d'(in)égalité entre les données des nœuds soient préservées dans le sous-arbre de v de profondeur d. Le sous-arbre de v de profondeur $d - 1$ contient moins de r^d nœuds, on peut toujours faire le recoloriage à l'aide des m classes. On appelle t_{rc}, ce nouvel arbre. Recolorier les données de l'arbre ne change rien à l'appartenance de sa projection sur l'alphabet fini au langage L. De plus, comme on ne touche pas aux classes des positions marquées, le nouvel arbre contient toujours les témoins précédemment marqués. Enfin, comme les patterns d'arbre négatifs ne contiennent que des relations $\mid$ entre les nœuds et sont de profondeur inférieure à d, l'arbre t_{rc} ne contient toujours pas de témoins pour les patterns d'arbre négatifs. Le nouvel arbre est donc toujours un modèle de la formule φ dont la projection sur l'alphabet fini appartient au langage L.

Cela conclut la première partie de la preuve.

On peut voir les modèles d'arité bornée par r contenant au plus m classes comme des arbres étiquetés sur un alphabet fini $\Sigma \times M$ où M est un alphabet de m lettres chacune correspond à une classe différente.

À partir d'un automate $\mathcal{A}_L$ qui reconnaît le langage L, on peut construire un automate fini $\mathcal{A}_{\varphi L}$ sur l'alphabet $\Sigma \times M$ qui reconnaît exactement les modèles

[12] L'arité peut être bornée par $N + K$ où N est la somme du nombre de nœuds des patterns de la formule et K est le maximum sur l'ensemble des transitions de $\mathcal{A}$ de la somme de $k + 1$ pour chaque contrainte $\#q \leq k$ (ou $\#q < k$) apparaissant dans la transition. Pour plus de détails, on réfère le lecteur à la preuve de la Proposition 4.21.

de φ qui respectent le langage L dont l'arité est bornée par r et qui contiennent au plus m classes que l'on va étiqueter avec les lettres de M. L'idée est de stocker dans un état de l'automate $\mathcal{A}_{\varphi L}$ un état de $\mathcal{A}_L$ et un arbre de profondeur d et d'arité au plus r étiqueté par $\Sigma \times M$. Cet automate est de taille doublement exponentielle dans la taille de φ et L et peut être construit en temps doublement exponentiel. Tester si une transition est valide correspond à un problème de model checking de patterns d'arbre sur un arbre où les patterns et l'arbre sont d'arité maximale r et de profondeur au plus p. Le vide d'un tel automate est PTime.

Le problème de satisfaisabilité de BC($\sim, |$) en présence de contrainte régulières est donc 2ExpTime. $\square$

Lemme 4.29 Le problème de satisfaisabilité de BC($\sim, |$) en présence de contrainte régulière est AExpSpace-dur.

Preuve. Cette borne inférieure s'obtient par la réduction à notre problème du problème d'acceptance des machines de Turing alternantes qui s'exécutent en espace exponentiel. Comme les classes de complexité AExpSpace et 2ExpTime sont égales [CKS81], on obtient ainsi la borne inférieure du Théorème 4.27. On commence par expliquer comment coder l'exécution d'une machine de Turing d'espace exponentiel dans un arbre de données puis on explique comment étendre ce codage aux machines de Turing alternantes.

On considère une machine de Turing $\mathcal{M}$ qui s'exécute en espace 2^n où n est la taille de l'entrée et une entrée w de taille n. La structure du codage d'une exécution est la même que dans la preuve d'indécidabilité du Théorème 4.17. Une exécution (voir la Figure 4.15) est représentée par un arbre de données contenant une branche étiquetée par la suite des transitions de l'exécution. Chaque nœud de cette branche contient un sous-arbre qui représente la configuration courante. La taille d'une configuration est au plus 2^n, on code la configuration par un $\mathcal{C}$-sous-arbre de la même façon que dans la preuve du Théorème 4.24.

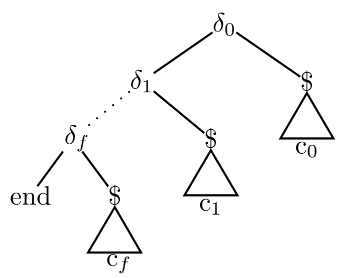


Fig. 4.15 –

On peut construire un langage régulier et une formule de patterns de BC($\sim, |$) tels que les modèles de la formule de patterns qui respectent le langage sont exactement le codage des exécutions acceptantes de la machine $\mathcal{M}$ sur l'entrée w.

Le langage va contraindre la structure finie de l'arbre. En particulier, le langage garantit que la suite de transitions respecte les règles de la machine (suite des états, configuration initiale et finale). Comme dans la preuve du Théorème 4.24, on utilise le fait que les patterns d'arbre peuvent comparer les positions d'un arbre de profondeur polynomiale. Afin de contraindre la structure de données des $\mathcal{C}$-sous-arbres, on utilise des patterns d'arbre similaires aux patterns présentés dans les Figures 4.11 et 4.12 ; on considère ces patterns où l'on a remplacé les relations $\|$ par des relations $|$. De la même manière, on peut distinguer des positions consécutives dans un $\mathcal{C}$-sous-arbre à l'aide de patterns similaires à ceux qui sont décrits dans la Figure 4.13(a). Deux $\mathcal{C}$-sous-arbres consécutifs correspondent aux sous-arbres des deux *nœuds* consécutifs étiquetés par des configurations elles aussi consécutives[13]. On peut alors construire des patterns d'arbre qui représentent les comportements indésirables entre deux configurations successives. On interdit ces patterns afin de garantir l'évolution correcte des configurations.

Cette construction s'étend de manière naturelle aux machines de Turing alternantes AExpSpace. Le codage d'une exécution est alors un arbre étiqueté par la suite des transitions ; si une transition correspond à un branchement universel, le *nœud* correspondant a deux enfants étiquetés par des transitions de la machine ; si une transition correspond à un branchement existentiel, le nœud correspondant a un enfant étiqueté par une transition. De plus chacun des nœuds contient un $\mathcal{C}$-sous-arbre représentant la configuration correspondante de la machine. Le langage régulier ainsi que la formule de patterns se construisent de la même manière que pour le cas précédent. Le langage régulier permet de garantir la structure finie de l'arbre (en particulier que la suite de transitions respecte les règles de la machine). La formule de patterns garantit la structure de donnée de l'arbre et assure que l'évolution entre deux $\mathcal{C}$-sous-arbres successifs est correcte. $\square$

Remarque 4.30 Comme dans la preuve du Théorème 4.17, les bornes inférieures des deux théorèmes précédents s'obtiennent en utilisant seulement une conjonction de négations de patterns d'arbre.

On déduit de cette remarque le résultat suivant :

Théorème 4.31 *Le problème de satisfaisabilité de* $BC^{-}(\sim, \|)$ *en présence de contrainte régulière est* 2ExpTime*-complet.*

4.3.3 Arbres de profondeur bornée

Dans cette partie on s'intéresse au problème restreint aux arbres de profondeur bornée. Borner la taille des modèles considérés permet d'obtenir la décidabilité du fragment général $\mathrm{BC}(\sim, \|, |)$. Formellement le problème considéré est le suivant :

Problème 4.32 Étant donnée une formule φ d'une logique F, un entier d et un langage régulier L, le problème de **satisfaisabilité à profondeur bornée en présence de contrainte régulière** est de savoir s'il existe un modèle de la formule φ de profondeur inférieure ou égale à d dont la projection sur l'alphabet fini appartient au langage L.

[13] On peut facilement parler de deux nœuds de ce type dans un pattern à l'aide du prédicat $|$.

Du point de vue de la complexité, on distingue deux problèmes : soit la borne d est fixée, soit elle fait partie des données du problème au même titre que la formule de patterns et le langage régulier. Dans les deux cas, on suppose que la borne d est codée en unaire.

Théorème 4.33 *Si d est fixée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour BC($\sim, \|, |$) est Σ_2-complet.*

Les deux lemmes suivants donnent les bornes supérieures et inférieures.

Lemme 4.34 Si d est fixée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour BC($\sim, \|, |$) est Σ_2.

Preuve. Soit un langage régulier L et une formule φ. À l'aide d'un argument de pompage identique à celui de la preuve du Théorème 4.27, on montre que s'il existe un modèle, il en existe un de taille polynomiale dans la taille de l'entrée.

On considère un arbre de données t qui satisfait la formule et respecte le langage. On divise les patterns d'arbre de φ en deux groupes : on appelle patterns d'arbre positifs les patterns d'arbre qui apparaissent dans l'arbre t et patterns d'arbre négatifs, ceux qui n'apparaissent pas. On marque dans l'arbre les nœuds d'un témoin de chacun des patterns d'arbre positifs. On peut ensuite pomper dans l'arbre en évitant les positions marquées et en respectant l'appartenance au langage L afin d'obtenir un modèle d'arité bornée. La profondeur de cet arbre étant par définition bornée par une constante, on a donc un arbre de taille polynomiale dans la taille de l'entrée φ et L.

L'algorithme suivant résoud donc le problème et appartient à la classe de complexité Σ_2 : (1) deviner le modèle ainsi qu'un témoin pour chaque pattern d'arbre positif et vérifier que les témoins sont corrects (2) vérifier que les patterns d'arbre négatifs n'apparaissent pas dans l'arbre (problème co-NP). $\square$

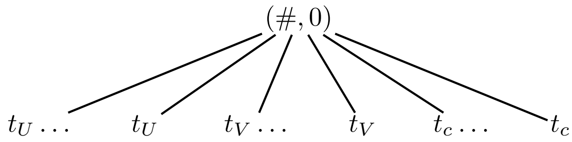
Lemme 4.35 Si d est fixée (> 1), le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour BC($\sim, \|, |$) est Σ_2-dur.

Preuve. Pour prouver ce résultat de dureté, on réduit le problème QSAT2 à notre problème de satisfaisabilité. Le Problème QSAT2 est le problème de satisfaisabilité pour les formules propositionnelles quantifiées de la forme $\exists U \forall V \theta$ où θ est une formule propositionnelle en forme 3-DNF.

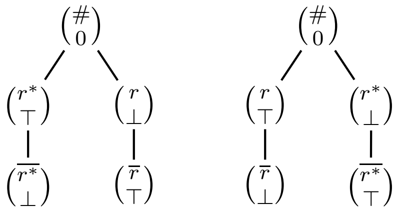
On considère une formule $\exists U \forall V \theta$ où θ est en forme 3-DNF. On va construire un ensemble d'arbres de données S et un pattern d'arbre ${P_\theta}^U$ tels que : un des arbres de S ne vérifie pas le pattern d'arbre ${P_\theta}^U$ si et seulement si la formule $\exists U \forall V \theta$ est satisfaisable. On décrit l'ensemble S comme l'ensemble des modèles d'une formule de patterns qui respectent une langage régulier donné.

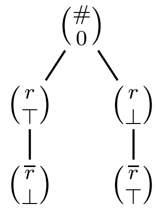
On décrit ici l'ensemble S et le pattern d'arbre ${P_\theta}^U$.

Intuitivement, chaque arbre de donnée de l'ensemble S correspond à la formule θ dont on a fixé la valuation des variables de l'ensemble U. La structure des arbres de S est proche de celle de l'arbre t_θ utilisé pour le codage du problème 3SAT dans le preuve de la Proposition 4.6. L'alphabet fini est $\Sigma = \{r, \overline{r}, r^*, \overline{r}^*, X, Y, Z, \#, \$\}$. Un arbre de S contient exactement trois classes que l'on identifiera aux valeurs de données 0, $\top$ et $\bot$ pour plus de lisibilité. Il contient aussi toutes les valuations possibles pour chaque variable. On considère dans le problème QSAT2 une formule θ en forme normale disjonctive ; un arbre contient aussi, pour chaque clause, toutes les valuations des littéraux qui la rendent fausse. De plus, on impose que chaque arbre de données de S contienne le marquage d'une valuation des variables de U. La structure de ces arbres est décrite dans la Figure 4.16.

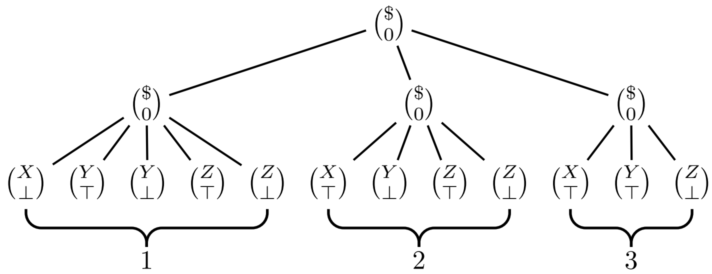


(a) La forme des arbres de S





(b) Les deux sous-arbres possibles t_U (c) Le sous-arbre t_V



(d) Le sous-arbre t_c

FIG. 4.16 – Les arbres de S

Les sous-arbres t_V sont identiques aux sous-arbres t_v de la preuve de la Proposition 4.6. Les sous-arbres t_U sont quasiment identiques. La seule différence réside dans les étiquettes : une des deux branches de l'arbre (correspondant à une valuation de la variable représentée) est marquée. Une clause de la formule θ est la conjonction de trois littéraux que l'on nommera respectivement X, Y, Z. Les arbres t_c sont très semblables à ceux du codage de 3SAT. Chaque sous-arbres t_c est composé de trois sous-arbres (Voir Figure 4.16), chacun d'entre eux représente maintenant une des trois possibilités de rendre une clausse fausse : soit (1) le premier littéral X est faux, soit (2) X est vrai mais Y est faux, soit

(3) X et Y sont vrais et Z est faux. On peut remarquer que les arbres de S sont de profondeur trois.

Le pattern d'arbre ${P_\theta}^U$ est quasiment identique au pattern d'arbre P_θ. On ne le décrit donc pas entièrement. Ce pattern d'arbre sélectionne une valuation par variable et une valuation par clause. La seule différence est dans l'étiquetage des nœuds qui force à choisir la valuation marquée pour chaque variable de U. Les contraintes de données garantissent la cohérence entre la valuation des variables et celle des clauses exactement de la même manière que dans le pattern d'arbre P_θ. Intuitivement, le pattern d'arbre décrit une valuation de variables V telle que la formule θ est fausse avec cette valuation pour les variables de V et la valuation marquée pour les variables de U.

On a la propriété suivante : "Un arbre de S satisfait le pattern d'arbre ${P_\varphi}^U$ si et seulement si la valuation marquée des variables de U n'est pas une solution de cette instance de QSAT2".

L'ensemble des arbres de S peut facilement être décrit comme modèles d'une formule de patterns et d'un langage régulier. Le langage assure la structure de l'arbre et les étiquettes finies. Tous les arbres de S ont la même forme. On peut représenter la structure de données à l'aide d'un pattern d'arbre qui décrit les relations d'(in)égalité de données entre les nœuds à l'aide des contraintes de données.

En résumé, étant donnée une instance du problème QSAT2 ($\exists U \forall V \theta$), on peut construire un langage régulier et une formule de patterns tels que la formule de patterns est satisfaisable par un arbre de données qui respecte le langage ssi l'instance de QSAT2 a une solution. La formule est la conjonction du pattern d'arbre qui décrit la structure de données des arbres de S et de la négation du pattern d'arbre ${P_\theta}^U$.

La preuve utilise des arbres de profondeur 3. En utilisant un codage similaire plus compact, on peut montrer le résultat pour des arbres de profondeur 2. $\square$

Théorème 4.36 *Si la borne d fait partie de l'entrée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour $BC(\sim, \|, |)$ est* NExpTime-*complet.*

Preuve. La preuve de NExpTime-dureté est identique à celle du Théorème 4.24.

La borne supérieure s'obtient à l'aide d'un argument de plus petit modèle. S'il existe un modèle, il en existe un de taille exponentielle dans la taille de l'entrée. On considère une formule de patterns φ, un langage régulier L et un entier d. Soit t un modèle de φ qui respecte L. On marque les positions d'un témoin pour chaque pattern de la formule de patterns qui apparaît dans t. En ignorant les valeurs de données, comme dans le cas d'un langage régulier d'arbre classique, on peut couper horizontalement l'arbre en respectant le langage L et sans couper une position marquée. On obtient ainsi un nouveau modèle de la formule qui respecte le langage L d'arité polynomiale dans la taille de l'entrée. La profondeur de l'arbre est par définition bornée par d. On a donc un modèle de taille exponentielle dans la taille de l'entrée.

Comme dans la preuve du Théorème 4.24, on en déduit un algorithme simple pour résoudre le problème de satisfaisabililté en NExpTime. Il commence par deviner un modèle de taille exponentielle, puis (1) vérifie pour chaque pattern d'arbre de la formule, si l'arbre contient ou non un témoin, (2) évalue le Parse

Tree de la formule et (3) vérifie que la projection sur l'alphabet fini de l'arbre appartient au langage L. $\square$

Remarque 4.37 Comme dans le cas de la réduction de 3SAT de la Proposition 4.6, les preuves des résultats de dureté peuvent être faites avec des patterns d'arbre utilisant des arêtes d'un seul type enfant ou descendant.

On déduit de cette remarque les résultats suivants :

Proposition 4.38 Si d est fixée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour $\mathrm{BC}(\sim, \|)$ ou $\mathrm{BC}(\sim, |)$ est Σ_2-complet.

Proposition 4.39 Si d fait partie de l'entrée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour $\mathrm{BC}(\sim, \|)$ ou $\mathrm{BC}(\sim, |)$ est NExpTime-complet.

Remarque 4.40 La preuve de NExpTime-dureté du Théorème 4.36 étant la même que celle du Théorème 4.24, la remarque concernant l'utilisation du seul fragment négatif est toujours valable.

On a donc le résultat suivant :

Proposition 4.41 Si d fait partie de l'entrée, le problème de satisfaisabilité à profondeur bornée en présence de contrainte régulière pour $\mathrm{BC}^-(\sim, |, \|)$, $\mathrm{BC}^-(\sim, \|)$ ou $\mathrm{BC}^-(\sim, |)$ est NExpTime-complet.

4.4 Récapitulatif des résultats pour les arbres

Voici un tableau récapitulatif des principaux résultats obtenus dans cette partie. La colonne *bnd* Sat correspond au problème de satisfaisabilité en présence de contrainte régulière pour des arbres de profondeur bornée où la borne fait partie de l'entrée du problème. La colonne bnd_f Sat correspond au cas où la borne est fixée.

Fragments	Model-Checking	Satisfaisabilité	*bnd* Sat	bnd_f Sat
$\mathrm{BC}(\sim, \|, |)$	$\mathrm{P}^{\|\mathrm{NP}}$ et BH-dur	Indécidable	NExpTime-complet	Σ_2-complet
$\mathrm{BC}(\sim, |)$	$\mathrm{P}^{\|\mathrm{NP}}$ et BH-dur	2ExpTime-complet	NExpTime-complet	Σ_2-complet
$\mathrm{BC}(\sim, \|)$	$\mathrm{P}^{\|\mathrm{NP}}$ et BH-dur	NExpTime-complet	NExpTime-complet	Σ_2-complet
$\mathrm{BC}^-(\sim, \|, |)$	coNP-complet	Indécidable	NExpTime-complet	Σ_2-complet
$\mathrm{BC}^+(\sim, \|, |)$	NP-complet	NP-complet	NP-complet	NP-complet
$Ptn(\sim, \|, |)$	NP-complet	PTime	PTime	PTime

Chapitre 5

La restriction aux mots de données

Dans cette partie, on s'intéresse au cas des mots de données. Un mot est un arbre d'arité un. Les patterns de mots peuvent être définis de la même manière sur ces structures. Les résultats de décidabilité sont les mêmes que dans le cas des arbres mais certains résultats de complexité sont différents du fait de la structure linéaire des mots.

5.1 Définition

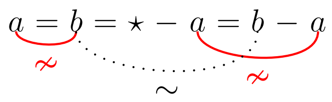
La définition d'un pattern pour les mots de données est la même que dans le cas des arbres mais on se restreint à des structures linéaires. On adopte dans cette partie un vocabulaire plus spécifique aux mots. On parle de positions au lieu de parler de nœuds et de relation successeur/ordre au lieu des relations enfant/descendant.

Définition 5.1 Un **pattern de mot** $P = (p, \mathcal{C}_\sim, \mathcal{C}_\nsim)$ est formé de :

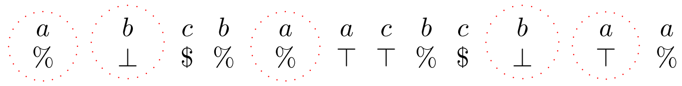
- un arbre d'arité 1 p
  - dont les positions sont étiquetées soit par une lettre de l'alphabet fini Σ, soit par le symbole wildcard $\star$
  - dont les arêtes entre deux positions successives sont étiquetées soit par $+1$ (relation successeur) soit par $<$ (relation ordre)
- deux relations binaires $\mathcal{C}_\sim$ et $\mathcal{C}_\nsim$ sur l'ensemble des positions de p.

La sémantique est définie de la même manière qu'avant : un mot de données w satisfait un pattern de mot $P = (p, \mathcal{C}_\sim, \mathcal{C}_\nsim)$ (noté $t \models P$) s'il existe une fonction injective f des positions de p vers les positions de w qui respecte les contraintes d'étiquettes, les contraintes de positions (successeur/ordre) ainsi que les contraintes d'(in)égalité de données entre les positions. Cette fonction est appelée témoin du pattern de mot dans le mot de données.

Pour des questions de mise en page et de lisibilité, on représentera les patterns de mots horizontalement en notant la relation d'ordre entre deux positions successives par $=$ et la relation successeur par $-$. Un exemple est donné dans la Figure 5.1.



(a) Un pattern de mot



(b) Un mot de données dans lequel les positions entourées constituent un témoin du pattern de mot

FIG. 5.1 – Exemples

5.2 Le problème de model-checking

La quasi totalité des résultats de complexité sont les mêmes que dans le cas des arbres. Le model-checking est toujours NP-complet si on considère un seul pattern de mot et dans le cas général le problème est BH-dur et dans $\mathrm{P}^{\|\mathrm{NP}}$. Bien sûr dans ces cas les bornes supérieures sont des conséquences du cas des arbres. En revanche, les résultats de dureté ne sont plus valables. On commence par montrer la NP-dureté du model checking d'un pattern de mot. Le résultat de dureté dans le cas général est alors obtenu de la même manière que dans le cas des arbres. On explicite ensuite la complexité du fragment successeur (correspondant au fragment enfant dans le cas des arbres). Le problème de model-checking dans ce cas est PTIME. C'est le seul fragment pour lequel la complexité du problème est différente.

Proposition 5.2 Le model checking d'un pattern de mot de $Ptn(\sim,<)$ est NP-dur.

Preuve. Comme dans le cas des arbres, on réduit le problème 3SAT à notre problème. L'idée de la réduction est exactement la même que dans le cas des arbres mais la structure linéaire des mots impose un codage légèrement différent. Étant donné une formule propositionnelle θ en forme 3CNF avec k variables et n clauses, on construit un mot de données w_θ et un pattern de mot P_θ (le tout en temps polynomial dans la taille de θ) tel que $w_\theta \vDash P_\theta$ si et seulement si θ est satisfiable.

Comme dans le codage précédent, le mot w_θ contient toutes les valuations possibles des variables et pour chaque clause les valuations des littéraux qui rendent la clause vraie. La structure du pattern de mot sélectionne une valuation de chaque variable et une valuation des littéraux pour chaque clause. Les contraintes de données assurent que la valuation choisie pour chaque littéral correspond bien à la valuation choisie pour les variables.

Plus précisément le mot w_θ est un mot de données de l'alphabet $\Sigma = \{r, \overline{r}, X, Y, Z, \#, \$\}$ qui contient exactement six classes que l'on désignera par les données 0,1,2,3,$\top$ et $\perp$ pour plus de clarté. Le mot w_θ est constitué de k copies d'un mot u_v suivies de n copies d'un mot u_c.

$$w_\theta = \underbrace{u_v \ \cdots \ u_v}_{k \text{ fois}} \underbrace{u_c \ \cdots \ u_c}_{n \text{ fois}}$$

(a) Le mot u_θ

$$u_v = \begin{matrix} \# & r & \overline{r} & r & \overline{r} \\ 0 & \top & \bot & \bot & \top \end{matrix}$$

(b) Le mot u_v

$$u_c = \begin{matrix} \# \\ 0 \end{matrix} \underbrace{\begin{matrix} \$ & X & Y & Y & Z & Z & \$ \\ 1 & \top & \top & \bot & \top & \bot & 1 \end{matrix}}_{1} \underbrace{\begin{matrix} \$ & X & Y & Z & Z & \$ \\ 2 & \bot & \top & \top & \bot & 2 \end{matrix}}_{2} \underbrace{\begin{matrix} \$ & X & Y & Z & \$ \\ 3 & \bot & \bot & \top & 3 \end{matrix}}_{3} \begin{matrix} \# \\ 0 \end{matrix}$$

(c) Le mot u_c

FIG. 5.2 – Le codage d'une formule θ contenant k variables et n clauses.

Comme dans le cas des arbres, un mot u_v représente les deux valuations possibles d'une variable. Un mot u_c représente toutes valuations des littéraux qui rendent une clause vraie. Une clause est la disjonction de trois littéraux représentés par les lettres X, Y, et Z. Le mot u_c (représenté dans la Figure 5.2) est composé de trois parties qui représentent ces valuations : pour qu'une clause soit vraie, (1) soit le premier littéral est vrai, (2) soit le premier littéral X est faux mais le second Y est vrai, (3) soit les deux premiers littéraux sont faux et le dernier est vrai. On décrit maintenant le pattern de mot $P_\theta = (p_\varphi, \mathcal{C}_\sim, C_\nsim)$.

La relation entre les positions du pattern de mot sera toujours la relation d'ordre, on ne le précise donc pas dans la suite. À nouveau p_θ est composé de k occurrences du mot $p_v = \#r\overline{r}$ suivies de n occurrences du mot $p_c = \#\$XYZ\$$. Les contraintes de données sont de deux types. Une partie des contraintes permettent d'assurer que le pattern de mot sélectionne bien exactement une valuation de chaque variable et une valuation des littéraux de la clause qui la rend vraie.

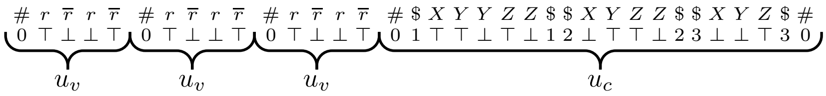
- Pour chaque p_v, on demande que les positions r et $\overline{r}$ soient dans $C_\nsim$.
- Pour chaque p_c, on impose que les deux positions \$ soient dans $C_\sim$.

Le reste des contraintes assurent la cohérence entre les valuations des variables et celles des littéraux. L'idée est identique à celle utilisée dans le cas des arbres. Supposons que le premier littéral de la clause c est la variable x (resp. la négation de la variable x). Alors, on ajoute à la relation $C_\sim$ la paire formée de la position r (resp. $\bar{r}$) du mot p_v correspondant à la variable x et de la position X du mot p_c correspondant à la clause c. On fait la même chose avec les autre littéraux de la clause correspondant à des positions Y et Z. La Figure 5.3 donne un exemple de la réduction pour la formule θ constituée de l'unique clause $a \vee \neg b \vee c$.

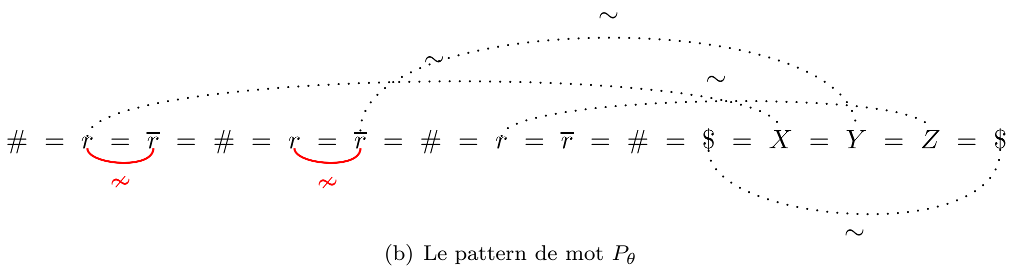
Il reste à montrer que $w_\theta \vDash P_\theta$ si et seulement si θ est satisfaisable. La preuve de ce résultat est exactement la même que dans le cas des arbres. $\square$

Remarque 5.3 Dans le cas des arbres, la réduction du problème 3SAT pouvait se faire en utilisant indifféremment un pattern d'arbre de $Ptn(\sim, |)$ ou $Ptn(\sim, \|)$. Ce n'est plus le cas si on considère les mots de données. L'utilisation du prédicat d'ordre est important. On va montrer dans la suite que le model-checking se fait en temps polynomial si on interdit ce prédicat dans les patterns de mots.

À partir de ce codage de 3SAT, on obtient de la même façon que dans le cas des arbres les résultats suivants.



(a) Le mot w_θ



(b) Le pattern de mot P_θ

FIG. 5.3 – Exemple : $\theta := a \vee \neg b \vee c$

Théorème 5.4 *Le model-checking est* NP*-complet pour* $Ptn(\sim, <, +1)$*,* $Ptn(\sim, <)$*,* $BC^+(\sim, <, +1)$ *et* $BC^+(\sim, <)$*.*

Théorème 5.5 *Le model-checking est co*NP*-complet pour la négation d'un pattern de mot de* $Ptn(\sim, <, +1)$ *ou* $Ptn(\sim, <)$ *et pour* $BC^-(\sim, <, +1)$ *et* $BC^-(\sim, <)$*.*

Théorème 5.6 *Le model-checking est* BH*-dur pour* $BC(\sim, <, +1)$ *et* $BC(\sim, <)$*.*

Le résultat qui diffère du cas des arbres est la complexité du problème pour le fragment successeur (correspondant au fragment enfant des arbres).

Théorème 5.7 *Le model-checking de* $BC(\sim, +1)$ *est* PTIME*.*

Preuve. L'idée de la preuve est qu'un pattern de mot qui utilise uniquement le prédicat successeur ne peut apparaître que dans une fenêtre de longueur bornée. Si on considère un mot de données et un pattern de mot de $Ptn(\sim, +1)$ on peut tester si le pattern de mot apparait dans le mot en parcourant le mot de gauche à droite. À chaque étape, on teste si le pattern apparaît dans une fenêtre de la taille du pattern de mot. Étant donné une formule de patterns de $\mathrm{BC}(\sim, +1)$, on fait ce test pour chacun des patterns de mots de la formule, on construit le Parse Tree de la formule et on l'évalue. Cet algorithme est polynomial. □

5.3 Le problème de satisfaisabilité

Considérer le cas des mots de données ne change pas les résultats de décidabilité. Au niveau de la complexité, la structure linéaire des arbres permet de diminuer la complexité de certains fragments.

5.3.1 Indécidabilité

Ces résultats ne sont pas une conséquence de ceux des arbres bien qu'il soient identiques. On prouve les résultats en adaptant aux mots les preuves utilisées dans le cas des arbres.

Théorème 5.8 *Le problème de satisfaisabilité pour BC*($\sim, +1, <$) *en présence de contrainte régulière est indécidable.*

Preuve. Le schéma de la preuve est exactement le même que dans le cas des arbres. On réduit le problème d'arrêt des machines à deux compteurs à notre problème. L'idée du codage est similaire au cas des arbres mais diffère du fait de la structure linéaire des mots.

Étant donnée une machine $\mathcal{M}$, on construit un langage régulier $L_{\mathcal{M}}$ et une formule de patterns $\varphi_{\mathcal{M}}$ tels que les modèles de la formule de patterns dont la projection sur l'alphabet fini appartient au langage régulier correspondent aux codages des exécutions acceptantes de la machine.

Dans la suite, on décrit le codage d'une exécution d'une machine $\mathcal{M}$ en un mot de données puis on décrit le langage $L_{\mathcal{M}}$ et la formule de patterns $\varphi_{\mathcal{M}}$.

Le codage d'une exécution peut être découpé en trois parties :

1. La forme générale du mot qui va être décrite dans le langage régulier.
2. La structure de données propre à chaque configuration.
3. L'évolution de la structure de données entre deux configurations successives.

Le codage d'une exécution est constitué d'une suite de mots de données correspondant à la suite des transitions et configurations de l'exécution. La projection sur l'alphabet fini de chacun de ces sous-mots est de la forme $\#\$\delta(\$a)^+(\$b)^+\$\#$. Il contient une transition de la machine ainsi que les valeurs des deux compteurs codées en unaire par les positions a et b[1]. Les positions étiquetées # et \$ sont présentes pour des raisons techniques. La structure générale du mot est contrôlée par le langage régulier. Il assure que le mot est bien une suite de configurations codée de la manière décrite précédemment. Il permet aussi de vérifier que la suite des transitions respecte les règles de la machine $\mathcal{M}$ (suite d'états et configurations initiale et finale). Il vérifie aussi le résultat des tests à zéro (qui correspond à la présence d'un seul[2] a ou de b dans une configuration).

$$u_{cf_0}u_{cf_1}u_{cf_2}\cdots u_{cf_n}$$

(a) La codage d'une exécution

$$\begin{array}{cccccccccccccccccccc}
\# & \$ & \delta_i & \$ & a & \$ & a & & \$ & a & \$ & b & \$ & b & & \$ & b & \$ & \# \\
\gamma & \gamma & & \gamma & \alpha_1 & \gamma & \alpha_2 & \cdots & \gamma & \alpha_k & \gamma & \beta_1 & \gamma & \beta_2 & \cdots & \gamma & \beta_m & \gamma & \gamma
\end{array}$$

(b) Le codage d'une configuration u_{cf_i}

FIG. 5.4 – Codage d'une exécution

Les données permettent de contrôler l'évolution de la valeur des compteurs entre deux configurations successives. L'idée est la même que dans le cas des arbres. On ajoute simplement une contrainte sur les positions # et \$ afin de distinguer les positions d'une même configuration. C'est inutile dans le cas des arbres où chaque configuration correspond à un sous-arbre.

1. Les positions \$ et # d'une même configuration sont toutes dans la même classe et cette classe est différente pour chaque configuration.

[1] Pour des raisons techniques on a ajoute toujours un a et un b à la valeur réelle des compteurs

[2] cf note précédente.

2. Au niveau structure d'une configuration, chaque a (resp. b) est dans une classe différente.
3. Si deux positions a correspondent toutes les deux au k-ième a dans deux configurations successives, alors elles sont dans la même classe.

La formule de patterns $\varphi_{\mathcal{M}}$ assure les propriétés décrites ci-dessous en interdisant les patterns de mots associés (qui décrivent les comportements violant la propriété).

Une première série de propriétés assure que chaque configuration est associée à une valeur de donnée différente qui correspond à la classe des positions \$ et #. Cela va nous permettre dans la suite de reconnaitre les positions appartenant à une même configuration ainsi que les positions de deux configurations successives.

– Tous les # suivis d'un \$ et les # successeurs d'un \$ sont dans des classes différentes.

$$\# - \$ = \# - \$ \quad (\sim) \qquad \$ - \# = \$ - \# \quad (\sim)$$

– Les positions \$ et les # d'une même configuration sont toutes dans la même classe.

$$\# - \$ \quad (\not\sim) \qquad \$ - \star - \$ \quad (\not\sim) \qquad \$ - \# \quad (\not\sim)$$

Les autres propriétés imposées par la formule de patterns sont les mêmes que dans le cas des arbres :

– Dans une configuration, tous les a et les b sont dans des classes différentes.

$$\$ - a = \$ - a \quad (\sim \ \sim) \qquad \$ - b = \$ - b \quad (\sim \ \sim)$$

– Il y a une bijection entre les a (resp b) communs de deux configurations successives :
Interdire ce premier pattern de mot assure que les premiers a de deux configurations successives sont dans la même classe.

$$\# - \$ - \star - \$ - a = \# - \# - \$ - \star - \$ - a \quad (\sim,\ \not\sim)$$

Interdire ce second pattern assure la continuité de la propriété.

$$\$ - a - \$ - a = \# - \# = \$ - a \ \$ - a \quad (\sim,\ \sim,\ \sim,\ \not\sim)$$

– une transition d'incrémentation du premier compteur revient à rajouter un a à la configuration suivante :

Interdire ce pattern de mot assure qu'il y a pas de a dans la configuration suivante.

$$\$ - a = \# - \# = \$ - a - \$ - b$$

Interdire le pattern de mot suivant assure qu'il n'y a pas plus d'un a supplémentaire dans la configuration suivante.

$$\$ - a - \$ - b = \# - \# = \$ - a - \$ - a - \$ - a$$

– une décrémentation du premier compteur revient à faire disparaître un a Interdire ce pattern de mot assure qu'il y a moins de a dans la configuration suivante.

$$\$ - a - \$ - b = \# - \# = \$ - a$$

Interdire le pattern de mot suivant assure qu'il n'y a pas plus de deux a en moins dans la configuration suivante.

$$\$ - a - \$ - a - \$ - a = \# - \# = \$ - a - \$ - b$$

Nous avons maintenant tous les éléments de la réduction. La formule $\varphi_{\mathcal{M}}$ est la conjonction de la négation des patterns d'arbre décrits précédemment. Par construction les modèles de la formule $\varphi_{\mathcal{M}}$ qui respectent le langage régulier $L_{\mathcal{M}}$ sont exactement les codages des exécutions acceptantes de la machine $\mathcal{M}$.

Le problème de satisfaisabilité des combinaisons booléennes de patterns de mots est donc indécidable. □

Remarque 5.9

1. Comme dans le cas des arbres la formule de patterns utilisée dans la preuve du Théorème 5.8 est la conjonction de négations de patterns de mots.
2. On peut aussi représenter le langage utilisé dans cette même preuve à l'aide d'une formule de patterns qui utilise des patterns de mots positifs. Dans ce cas, il suffit d'enrichir le modèle en considérant un alphabet étendu à l'aide des états d'un automate qui reconnaît le langage régulier. À l'aide d'une disjonction de négation de pattern, on force chaque position à être marquée par exactement un état. A l'aide de deux patterns positifs, on force l'existence d'un état initial ainsi que d'un état final. À l'aide de négations de patterns, on peut forcer l'état initial à apparaître en première position et l'état final en dernière position. En interdisant tous

les comportements indésirables, on peut forcer l'étiquetage de tout couple de positions consécutives à respecter les transitions de l'automate.

On a donc aussi les résultats suivants :

Théorème 5.10 *Le problème de satisfaisabilité pour* $BC^{-}(\sim, +1, <)$ *en présence de contrainte régulière est indécidable.*

Théorème 5.11 *Le problème de satisfaisabilité pour* $BC(\sim, +1, <)$ *est indécidable même sans contrainte régulière.*

La décidabilité du problème de satisfaisabilité dans le cas des arbres nous donne la décidabilité dans le cas des mots. Mais du point de vue de la complexité, certains résultats sont à revoir.

5.3.2 Fragment positif

Proposition 5.12 Le problème de satisfaisabilité de $\mathrm{BC}^{+}(\sim, +1, <)$ en présence de contrainte régulière est NP-complet.

Preuve. La borne supérieure est une conséquence de la proposition 4.21.

La borne inférieure s'obtient comme dans le cas des arbres à partir du codage de 3SAT utilisé dans la preuve de la Proposition 5.2. Étant donné une formule propositionnelle θ, on choisit le langage constitué de l'unique mot correspondant à la projection sur l'alphabet fini du codage de la formule. On construit un pattern de mot à partir du codage en remplaçant les données par les contraintes correspondantes. La conjonction de ce pattern de mot et du pattern de mot P_θ forme une formule de patterns telle que : les modèles de cette formule et du langage régulier sont exactement les codages des valuations qui rendent la formule θ vraie. $\square$

Remarque 5.13 La borne inférieure peut déjà être obtenue dans le cas d'arbres sans données. C'est toujours le cas si on considère des mots. On peut adapter le codage proposé dans la preuve du Théorème 4.5 de [BFG05].

La preuve du Théorème 5.8 utilise de manière significative les relations ordre et successeur afin de compter la valeur des compteurs (a priori non-bornée). Il n'est donc pas surprenant que la restriction de l'utilisation de ces relations mène à la décidabilité.

5.3.3 Fragment Ordre

Théorème 5.14 *Le problème de satisfaisabilité en présence de contrainte régulière pour le fragment* $BC(\sim, <)$ *est* Σ_2*-complet.*

Preuve. On montre d'abord la borne supérieure. Un mot de données est un arbre avec une seule branche. D'après la preuve du Théorème 4.24, s'il existe un modèle, il en existe un de profondeur polynomiale. Donc dans le cas des mots, on en déduit que s'il existe un modèle, il en existe un de longueur polynomiale. Un algorithme Σ_2 pour résoudre le problème consiste donc à deviner le modèle de taille polynomiale ainsi que la valuation correspondante des patterns de la

formule puis vérifier ensuite qu'on a deviné juste. Il faut donc d'une part vérifier que la valuation est correcte rend la formule vraie (PTime) et que la valuation correspond au modèle (NP).
Cet algorithme est dans Σ_2.

La preuve de Σ_2-dureté suit la même idée que la preuve du Théorème 4.33. On réduit le problème QSAT2 à notre problème avec un codage similaire à celui utilisé dans la réduction de 3SAT de la preuve de la Proposition 5.2. Le Problème QSAT2 est le problème de satisfaisabilité pour les formules propositionnelles quantifiées de la forme $\exists U \forall V \theta$ où θ est un formule propositionnelle en forme 3-DNF.

On considère une formule $\exists U \forall V \theta$ où θ est en forme 3-DNF. On va construire un ensemble de mots de données S et un pattern de mot ${P_\theta}^U$ tel que un des mots de S ne vérifie pas le pattern ${P_\theta}^U$ si et seulement si la formule $\exists U \forall V \theta$ est satisfaisable. On décrit l'ensemble S comme l'ensemble des modèles d'une formule de patterns qui respectent une langage régulier donné.

On décrit ici l'ensemble S et le pattern de mot ${P_\theta}^U$.

Intuitivement, chaque mot de données de l'ensemble S corresponds à la formule θ dont on a fixé la valuation des variables de l'ensemble U. La structure des mots de S est proche de celle du mot w_θ utilisé pour le codage du problème 3SAT dans le preuve de la Proposition 5.2. L'alphabet fini est $\Sigma = \{r, \overline{r}, r^*, \overline{r}^*, X, Y, Z, \#, \$\}$. Un mot de S contient exactement six classes que l'on identifiera aux valeurs de données 0,1,2,3,$\top$ et $\bot$ pour plus de lisibilité. Il contient aussi toutes les valuations possibles pour chaque variable. On considère dans le problème QSAT2 une formule θ en forme normale disjonctive ; un mot de S contient aussi, pour chaque clause, toutes les valuation des littéraux qui la rendent fausse. De plus, on impose que chaque élément de S contienne le marquage d'une valuation des variables de U. La structure de ces mots de données est décrite dans la Figure 5.5.

$$u_U \cdots u_U \;\; u_V \cdots u_V \;\; u_c \cdots u_c$$

(a) La forme des mots de S

$$\begin{array}{ccccc} \# & \overline{r}^* & r^* & \overline{r} & r \\ 0 & \top & \bot & \bot & \top \end{array} \qquad \begin{array}{ccccc} \# & \overline{r} & r & \overline{r}^* & r^* \\ 0 & \top & \bot & \bot & \top \end{array} \qquad \begin{array}{ccccc} \# & \overline{r} & r & \overline{r} & r \\ 0 & \top & \bot & \bot & \top \end{array}$$

(b) Les deux mots u_U possibles

(c) Le mot u_V

$$u_c = \begin{array}{c} \# \\ 0 \end{array} \underbrace{\begin{array}{cccccccc} \$ & X & Y & Y & Z & Z & \$ \\ 1 & \bot & \bot & \top & \bot & \top & 1 \end{array}}_{1} \underbrace{\begin{array}{cccccc} \$ & X & Y & Z & Z & \$ \\ 2 & \top & \bot & \bot & \top & 2 \end{array}}_{2} \underbrace{\begin{array}{ccccc} \$ & X & Y & Z & \$ \\ 3 & \top & \top & \bot & 3 \end{array}}_{3} \begin{array}{c} \# \\ 0 \end{array}$$

(d) Le sous-mot u_c.

Fig. 5.5 – Les mots de S

Les sous-mots u_V sont identiques aux sous-mots u_v de la preuve de la Proposition 5.2. Les sous-mots u_U sont quasiment identiques. La seule différence réside dans les étiquettes : une des deux parties du mot (correspondant à une valuation de la variable représentée) est marquée. Une clause de la formule θ est la conjonction de trois littéraux que l'on nommera respectivement X, Y, Z. Les mots u_c sont très semblables à ceux du codage de 3SAT ; ils sont formés de trois parties, chacune d'entre elles représente maintenant une des trois possibilités de rendre une clause fausse : soit (1) le premier littéral X est faux, soit (2) X est vrai mais Y est faux, soit (3) X et Y sont vrais et Z est faux.

Le pattern ${P_\theta}^U$ est quasiment identique au pattern de mot P_θ. On ne le décrit donc pas entièrement. Ce pattern de mot sélectionne une valuation par variable et une valuation par clause. La seule différence est dans l'étiquetage des nœuds qui force à choisir la valuation marquée pour chaque variable de U. Les contraintes de données garantissent la cohérence entre la valuation des variables et celle des clauses exactement de la même manière que dans le pattern d'arbre P_θ. Intuitivement, le pattern décrit une valuation des variables V telle que la formule θ est fausse avec cette valuation pour les variables de V et la valuation marquée pour les variables de U.

On a la propriété suivante : "Un mot de S satisfait le pattern de mot ${P_\theta}^U$ si et seulement si la valuation marquée des variables de U n'est pas une solution de cette instance de QSAT2".

L'ensemble S peut être décrit comme l'ensemble des modèles d'une formule de patterns dont la projection sur l'alphabet fini appartient à un certain langage régulier[3]. Le langage assure la forme du mot et l'étiquetage des positions. Tous les mots de S ont la même forme. On peut représenter la structure de données à l'aide d'un pattern de mot qui décrit les relations d'(in)égalité de données entre les positions à l'aide des contraintes de données.

En résumé, étant donné une instance du problème QSAT2 ($\exists U \forall V \theta$), on peut construire un langage régulier et une formule de patterns tels que la formule de patterns est satisfaisable par un mot de données qui respecte le langage si et seulement si l'instance de QSAT2 a une solution. La formule est la conjonction du pattern de mot qui décrit la structure de données des éléments de S et de la négation du pattern de mot ${P_\theta}^U$. $\square$

5.3.4 Fragment Successeur

Quand on se restreint au cas du fragment successeur, le problème de satisfaisabilité en présence de contrainte régulière devient PSPACE-complet. C'est le résultat du Théorème 5.17. Il est intéressant de remarquer que le résultat est vrai aussi sans contrainte de données. De plus comme dans le cas des arbres ce sont les négations qui font la difficulté du problème. Le Lemme 5.16 donne le résultat de dureté pour une conjonction de négations de pattern de mot ne contenant pas de contrainte de données. Le Lemme 5.15 montre un résultat plus fort que la borne supérieure du Théorème 5.17 puisqu'il autorise aussi des contraintes d'ordre entre les valeurs de données comme dans la Section 6.1.2.

Lemme 5.15 Le problème de satisfaisabilité en présence de contrainte régulière pour BC($\sim$, +1) est PSPACE.

[3]Pour les curieux le langage régulier est le suivant (avec c le nombre de clauses) :

$$((\#\overline{r}^*r^*\overline{r}r) + (\#\overline{r}^*r^*\overline{r}r))^{|U|}(\#\overline{r}^*r^*\overline{r}r)^{|V|}(\#\$XYYZZ\$\$XYZZ\$\$XYZ\$\#)^c$$

On pourrait prouver ce résultat à l'aide une preuve similaire à celle du Théorème 4.27 basé sur un argument de recoloriage qui permet de borner le nombre de classes des modèles à considérer. Cependant on prouve ce résultat de manière légèrement différente afin d'avoir une preuve qui s'étende directement au cas discuté dans la partie 6.1.2.

Preuve. Étant donnés une formule de patterns de BC($\sim, +1$) et un langage régulier, on va montrer qu'il existe un automate fini de taille exponentielle dans la taille de l'entrée qui reconnaît la projection sur l'alphabet fini des modèles.

L'idée de la preuve est la suivante : le témoin d'un pattern de mot utilisant uniquement la relation successeur entre ses nœuds apparaît forcément dans une fenêtre de taille bornée d'un mot de données. Donc dans un mot, pour savoir si une position peut être le début d'un témoin d'un pattern donné, il suffit de regarder un sous-mot de longueur finie. Ce qui importe alors n'est pas la valeur de chacunes des positions de ce mot mais les valeurs relatives des données dans les positions d'un sous-mot de taille finie. On peut alors représenter ces valeurs relatives à l'aide d'un alphabet fini (on donne plus d'explications dans la suite).

On construit un automate fini $\mathcal{A}_{\varphi,L}$ qui reconnaît la projection sur l'alphabet fini de ce type de modèles. Cet automate est de taille exponentielle dans la taille de l'entrée. On peut calculer si une transition est valide en espace polynomial. Le vide d'un tel automate est NLogSpace. Comme NPSpace=PSpace, on obtient ainsi notre borne supérieure.

On considère un langage régulier L, une formule de patterns φ contenant n patterns. On appelle k la longueur du plus long pattern de mot de φ. On considère un alphabet fini de k lettres $\Sigma_{\mathcal{D}} = \{d_1, \cdots d_k\}$.

On considère un mot de données u de longueur k. Si du point de vue des données, on s'intéresse uniquement à leurs valeurs relatives les unes par rapport aux autres, on peut représenter ce mot à l'aide d'un mot[4] $\bar{u}$ étiqueté sur $\Sigma \times \Sigma_{\mathcal{D}}$ (Voir l'exemple de la Figure 5.6). Les deux mots ont la même projection sur l'alphabet Σ. Les classes de u sont représentées dans $\bar{u}$ par des lettres de $\Sigma_{\mathcal{D}}$, de telle sorte que les relations d'équivalence entre les lettres représentant les classes respectent les relations entre les données correspondantes. Étant donné un mot $\bar{u}$ étiqueté par $\Sigma \times \Sigma_{\mathcal{D}}$, l'ensemble des mots de données qui peuvent être représentés par $\bar{u}$ peut être décrit à l'aide d'un pattern de mot dont la structure finie est exactement la projection sur l'alphabet fini de u et les contraintes de données décrivent exactement les valeurs relatives des données décritent par les lettres de $\Sigma_{\mathcal{D}}$.

Un mot étiqueté par $\Sigma \times \Sigma_{\mathcal{D}}$ peut être vu de trois manières différentes (Voir l'exemple de la Figure 5.6) :

- un mot au sens classique du terme étiqueté par un alphabet fini.
- un mot de données étiqueté sur Σ en considérant la partie $\Sigma_{\mathcal{D}}$ comme des valeurs de données.
- un pattern de mot : le mot étiqueté sur Σ décrit la structure finie du pattern de mot (on considère que la relation entre deux positions successives est toujours la relation successeur), les étiquettes de $\Sigma_{\mathcal{D}}$ représentent les contraintes de données entre les positions ((in)égalité et ordre).

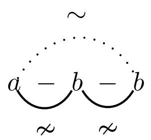
[4]Ce mot n'est pas forcément unique.

$u = \begin{smallmatrix} a & b & b \\ 2 & 7 & 2 \end{smallmatrix}$

(a) Le mot $u \in \Sigma \times \mathcal{D}$

$\bar{u} = \begin{smallmatrix} a & b & b \\ d_1 & d_2 & d_1 \end{smallmatrix}$

(b) un mot $\bar{u} \in \Sigma \times \Sigma_{\mathcal{D}}$



(c) Le pattern associé

FIG. 5.6 – Un exemple pour k=3

On considère un automate fini $\mathcal{A}_L = (Q, Q_i, Q_f, \delta)$ qui reconnaît le langage L. Comme d'habitude, Q représente l'ensemble des états, Q_i (resp. Q_f) les états initiaux (resp. finaux) et δ est la fonction de transition de l'automate.

Les états de l'automate $\mathcal{A}_{\varphi,L}$ sont les éléments de $Q \times (\Sigma \times \Sigma_{\mathcal{D}})^k \times \{0,1\}^n$. L'idée est de stocker dans un état de l'automate la représentation du mot de $\Sigma \times \Sigma_{\mathcal{D}}$ de longueur k de la position courante et la liste des patterns déjà rencontrés. La première partie représente un état de l'automate $\mathcal{A}_L$, la seconde partie représente un sous-mot de longueur k et le vecteur de bits final permet de garder en mémoire si on a déjà rencontré un témoin pour chacun des patterns de mots de la formule.

Une transition de l'automate $\mathcal{A}_{\varphi,L}$ est de la forme $(q, u, b), a \to (q', u', b')$ où :

- $q, a \to q'$ est une transition de $\mathcal{A}_L$
- le mot u' vu comme un mot de données qui satisfait le pattern de mot obtenu à partir du pattern décrit par le mot u dont on a tronqué la première position et auquel on a ajouté un a en dernière position sans ajouter de contrainte de données.
- le vecteur des bits b' est correct en fonction de b et de u'. Un bit de b' vaut 1 s'il valait déjà 1 dans b ou si le pattern de mot correspondant apparaît dans u'.

Le problème de savoir si une transition $(q, u, b), a \to (q', u', b')$ est une transition valide de l'automate $\mathcal{A}_{\varphi,L}$ est un problème PSPACE. En effet, vérifier si $q, a \to q'$ est une transition de $\mathcal{A}_L$ est trivial et vérifier si les couples (u, u') et (b, b') vérifient les propriétés précédentes revient à faire du model-checking ce qui est PSPACE d'après le Théorème 5.6.

Un état (q, u, b) est final si et seulement si q est final pour $\mathcal{A}_L$ et b représente une valuation des patterns de mots qui rend la formule φ vraie. Ceci se calcule en temps polynomial à partir de la formule à l'aide du Parse Tree. Cette étape est donc elle aussi PSPACE.

Par construction cet automate reconnaît la projection sur l'alphabet fini des modèles de φ qui respectent le langage régulier.

- Étant donné un modèle de φ qui respecte le langage régulier, on construit facilement une exécution acceptante de l'automate.
- Étant donné un mot accepté par l'automate $\mathcal{A}_{\varphi,L}$ et une exécution acceptante correspondante, on peut reconstruire un modèle de φ qui respecte L. Il suffit de respecter les contraintes de données indiquées par les mots témoins stockés dans les états de l'automate. Par définition des transitions et des états finaux, la projection sur l'alphabet fini du mot appartient à L et on peut choisir les données de sorte que le modèle vérifie la formule.

La taille de cet automate est exponentielle. Le vide d'un tel automate est NLogSpace [CDG+07] dans la taille de l'automate. On peut calculer à la volée les transitions de l'automate ainsi que les états acceptants avec une complexité PSpace. $\square$

Lemme 5.16 Le problème de satisfaisabilité en présence de contrainte régulière pour $BC^{-}(+1)$ est PSpace-dur.

Preuve. On prouve ce résultat en réduisant le problème d'acceptance des machines de Turing qui s'exécutent en espace polynomial à notre problème. On code une exécution de la machine dans un mot de données. On construit pour une machine et une entrée donnée un langage régulier et une formule de patterns dont les modèles sont exactement les exécutions acceptantes de la machine sur cette entrée.

On considère une machine de Turing $\mathcal{M}$ et une entrée w de taille n. La machine $\mathcal{M}$ s'exécute sur l'entrée w en espace m, polynomial en n. Une exécution acceptante de la machine sur w est une suite de transitions $\delta_0, \delta_1, \delta_2, \delta_3, \ldots, \delta_f$. De plus chaque configuration est de taille au plus m. On considère l'alphabet $\Sigma = \Gamma \cup Q \cup \Delta \cup \{\#\}$ où Γ représente l'alphabet du ruban, Q l'ensemble des états et Δ l'ensemble des transitions de la machine $\mathcal{M}$. On code une exécution de $\mathcal{M}$ dans un mot de la forme suivante :

$$\#u_0q_0\delta_0v_0\#u_1q_1\delta_1v_1\#\cdots\#u_fq_f\bot v_f \in (\#\Gamma^*Q\Delta\Gamma^*)^*$$

où chaque partie $\#u_iq_i\delta_iv_i\#$ est de longueur $m+4$ et représente une configuration de l'exécution telle que :

– le ruban de la machine contient le mot[5] $u_iv_i \in \Gamma^m$,
– la machine est dans l'état q_i,
– la prochaine transition de la machine est δ_i,
– les positions $q_i\delta_i$ marquent la position de la tête de lecture dans la machine.

On décrit maintenant le langage régulier et la formule de patterns tels que les modèles de la formule de patterns qui respectent le langage sont exactement le codage des exécutions acceptantes de la machine $\mathcal{M}$ sur l'entrée w.

Le langage régulier assure que la forme générale du mot est $(\#\Gamma^*Q\Delta\Gamma^*)^* \cap (\#\Sigma^{m+2})^*$. Il assure aussi que la suite des transitions et des états correspond bien à des transitions de la machine, que la première configuration représente une configuration initiale de la machine sur l'entrée w et que la dernière est bien une configuration finale.

La formule de patterns vérifie l'évolution du ruban et de la tête de lecture au cours de l'exécution. Le principe est toujours le même. La formule va interdire un certain nombre de patterns de mots. Chacun de ces patterns correspond à un comportement indésirable entre deux configurations successives. Dans un sous-mot correspondant à une configuration, on distingue deux types de positions : (1) les positions étiquetées par Γ qui ne sont pas voisines de la position étiquetée par Q ou de celle étiquetée par Δ et (2) les quatre positions restantes.

(1) L'étiquette des positions du premier type ne doit pas changer entre cette configuration et la suivante. Une telle position s'identifie facilement par le fait

[5] On suppose que l'alphabet Γ contient un symbole $\bot$ correspondant à une case vide avec lequel on peut compléter le contenu du ruban pour avoir toujours un mot de longueur m.

que ses voisins sont étiquetés par une lettre de Γ ou $\#$. La distance entre cette position et la position correspondante dans le configuration suivante est constante égale à $m+3$. On construit donc facilement les patterns de mots correspondant à un changement d'étiquette de ces positions comme on le montre dans la Figure 5.7. Les patterns de mots décrivent les situations où une position, étiquetée a et non voisine d'une position étiquetée par Q ou Δ, est étiquetée $c \neq a$ dans la configuration suivante. Le premier pattern de mot décrit un cas où la position a est la première position du ruban. Le second pattern de mot décrit le cas où la position a est la dernière position du ruban et le dernier pattern de mot décrit le cas où la position a est située ailleurs dans le ruban.

(2) L'évolution des quatre positions restantes dépend de la transition δ_i. Si la transition impose que la tête écrive a puis se déplace à droite, on interdit les patterns de mots décrits dans la Figure 5.8. Le premier pattern de mot décrit le cas où la position qui devrait contenir le a écrit par la machine contient une lettre b différente de a. Le second pattern de mot décrit la situation où la position de la nouvelle configuration qui devrait contenir la lettre b, qui suivait le tête de lecture dans la configuration de départ, contient une autre lettre de l'alphabet.

On remarque que la formule de patterns ne contraint pas l'évolution des positions étiquetées par un état ou une transition. Leur comportement est assuré par la combinaison du langage régulier et des patterns de mots qui force l'évolution des autres positions des configuration. Il serait donc redondant d'augmenter la taille de la formule de patterns.

$$\# - a - b - \underbrace{\star \cdots - \star}_{m+2 \text{ fois}} - c$$

(a) a,b,c sont des lettres de Γ et $a \neq c$

$$b - a - \# - \underbrace{\star \cdots - \star}_{m+2 \text{ fois}} - c$$

(b) a,b et c sont des lettres de Γ et $a \neq c$

$$d - a - b - c - \underbrace{\star \cdots - \star}_{m+2 \text{ fois}} - c$$

(c) a, b, c et d sont des lettres de Γ et $a \neq c$

FIG. 5.7 – Comportements interdits décrivant le cas (1)

$$\delta_i - \underbrace{\star \cdots - \star}_{m+1 \text{ fois}} - b$$

(a) Pour toute lettre b de Γ différente de a

$$\delta_i - b - \underbrace{\star \cdots - \star}_{m+3 \text{ fois}} - \alpha$$

(b) Pour toute lettres b de Γ et α de $\Sigma \backslash \{b\}$

FIG. 5.8 – Comportements interdits décrivant le cas (2) pour une transition qui écrit a et va à droite.

En tout, on a besoin d'un nombre polynomial de patterns interdits pour assurer l'évolution correcte des configurations dans le mot.

On a donc construit un langage régulier et une formule de patterns de taille polynomiale dans la taille de la machine $\mathcal{M}$ et de l'entrée w tels que les modèles de la formule de patterns qui respectent le langage régulier sont exactement le codage des exécutions acceptantes de la machine $\mathcal{M}$ sur l'entrée w. □

Théorème 5.17 *Le problème de satisfaisabilité de BC*($\sim, +1$) *en présence de contrainte régulière est* PSPACE*-complet.*

Preuve. La borne supérieure est un corollaire du Lemme 5.15.

La borne inférieure est un corollaire du Lemme 5.16. □

5.4 Récapitulatif des résultats pour les mots

Le tableau suivant rassemble les principaux résultats obtenus dans cette partie. Comparativement au cas plus général des arbres de données, les résultats de décidabilité sont les mêmes mais certains résultats de complexité sont différents du fait de la structure linéaire des mots par rapport à la structure branchante des arbres.

Fragments	Model-Checking	Satisfaisabilité
BC($\sim, <, +1$)	$\mathrm{P}^{\parallel\mathrm{NP}}$ et BH-dur	Indécidable
BC($\sim, +1$)	PTIME	PSPACE-complet
BC($\sim, <$)	$\mathrm{P}^{\parallel\mathrm{NP}}$ et BH-dur	Σ_2-complet
$\mathrm{BC}^-(\sim, <, +1)$	coNP-complet	Indécidable
$\mathrm{BC}^+(\sim, <, +1)$	NP-complet	NP-complet
Ptn($\sim, <, +1$)	NP-complet	PTIME

Chapitre 6

Discussion

6.1 Extensions

6.1.1 Plusieurs relations d'équivalence

On peut imaginer un modèle d'arbre de données plus général où chaque position contient plusieurs valeurs de données. Cette idée prend tout son sens dans le contexte des documents XML dont les éléments peuvent avoir plusieurs attributs que l'on peut représenter avec un tel modèle dans lequel chaque relation d'équivalence correspond à un attribut. Dans ce cadre, on peut définir de la même façon des patterns d'arbre avec plusieurs types de contraintes de données. Cela ne change rien à la complexité des problèmes considérés. Les preuves fonctionnent de la même manière. Ce résultat est surprenant si on le compare aux résultats obtenus pour les formalismes présentés dans la première partie de cette thèse. Le passage d'une à deux relations d'équivalence correspond à la frontière de décidabilité pour la logique FO^2 comme pour la logique temporelle munie de l'opérateur freeze.

6.1.2 Extension à l'ordre

Il est intéressant de considérer un ensemble de données avec plus de structure que le seul test d'inégalité. On a vu que dans le cas de la logique du premier ordre à deux variables, comme dans le cas de la logique temporelle munie de l'opérateur freeze, cela nous fait basculer dans l'indécidabilité. Dans le cas des combinaisons booléennes de patterns d'arbre, si on munit l'ensemble de données $\mathcal{D}$ d'une relation d'ordre, on peut augmenter le pouvoir expressif des patterns d'arbre en ajoutant la possibilité d'avoir des contraintes d'ordre entre les données des nœuds.

Dans ce contexte, les résultats de décidabilité comme de complexité restent inchangés dans le cas des mots comme dans celui des arbres. Les preuves sont identiques sauf dans le cas du Théorème 4.28. Dans ce cas, la preuve sans contrainte d'ordre sur les données utilise un argument de recoloriage des données afin de montrer qu'il existe un modèle avec un nombre borné de classe. Ce n'est plus le cas si on rajoute des contraintes d'ordre. On va donc donner une nouvelle preuve pour la borne supérieure de ce théorème.

Dans la suite, on note $\preccurlyeq$ cette relation d'ordre. Au niveau des figures, si deux nœuds x et y d'un pattern d'arbre sont dans la relation $C_{\preccurlyeq}$ (c'est à dire que l'on veut que $x \preccurlyeq y$), on représente cette contrainte par un arc fléché allant du nœud y au nœud x (voir l'exemple de la Figure 6.1).

Théorème 6.1 *Le problème de satisfaisabilité de BC($\sim, \preccurlyeq, |$) en présence de contrainte régulière est* 2ExpTime.

La nouvelle preuve suit la même structure et utilise les mêmes idées que la preuve du Théorème 5.15 dans le cas des mots.

Preuve. Étant donné une formule de patterns de BC($\sim, \preccurlyeq, |$) et un langage régulier, on va montrer qu'il existe un automate d'arbre fini de taille doublement exponentielle dans la taille de l'entrée qui reconnaît la projection sur l'alphabet fini des modèles dont l'arité est bornée.

L'argument de pompage horizontal utilisé dans les preuves précédentes nous assure que s'il existe un modèle, il en existe un d'arité polynomiale dans la taille de la formule de patterns et du langage.

L'idée de la preuve est la suivante : le témoin d'un pattern d'arbre utilisant uniquement des arêtes de type enfant apparaît dans une zone de profondeur bornée. Dans un arbre de données, pour savoir si un nœud peut être la racine d'un témoin d'un pattern d'arbre donné, il suffit de regarder un sous-arbre de profondeur bornée. Ce qui importe alors n'est pas la valeur de chacun des nœuds de ce sous-arbre mais les valeurs relatives des données dans les nœuds du sous-arbre. Si on considère des modèles d'arité bornée, le sous-arbre à considérer pour chaque nœud est de taille bornée. On peut alors représenter les données à l'aide d'un alphabet fini (plus d'explications dans la suite).

On construit un automate fini $\mathcal{A}_{\varphi,L}$ ascendant qui reconnaît la projection sur l'alphabet fini de ce type de modèles. Cet automate est de taille doublement exponentielle dans la taille de l'entrée et peut être construit en temps doublement exponentiel. Le vide d'un tel automate est PTime. On obtient ainsi notre borne supérieure.

On considère un langage régulier L, une formule de patterns φ contenant n patterns d'arbre. On appelle r la borne d'arité obtenue par l'argument de pompage précédent et d la profondeur maximale des patterns d'arbre de φ. On note $k = r^{d+1}$. On considère un alphabet fini de k lettres $\Sigma_{\mathcal{D}} = \{d_1, \cdots, d_k\}$ muni de la relation d'ordre naturelle ($d_i < d_j$ pour $i < j$).

On considère un arbre de données t d'arité r et de profondeur p. Si du point de vue des données, on s'intéresse uniquement à leurs valeurs relatives les unes par rapport aux autres, on peut représenter cet arbre à l'aide d'un arbre[1] $\bar{t}$ étiqueté sur $\Sigma \times \Sigma_{\mathcal{D}}$. Les deux arbres ont la même projection sur l'alphabet Σ. Les classes de t sont représentées dans $\bar{t}$ par des lettres de $\Sigma_{\mathcal{D}}$, de telle sorte que les relations d'ordre entre les lettres représentant les classes respectent les relations entre les données correspondantes. Étant donné un arbre $\bar{t}$ étiqueté par $\Sigma \times \Sigma_{\mathcal{D}}$, l'ensemble des arbres de données qui peuvent être représentés par $\bar{t}$ peut être décrit à l'aide d'un pattern d'arbre donc la structure d'arbre est exactement la projection sur l'alphabet fini de t et les contraintes de données représentent exactement les valeurs relatives des données décrites par les lettres de $\Sigma_{\mathcal{D}}$.

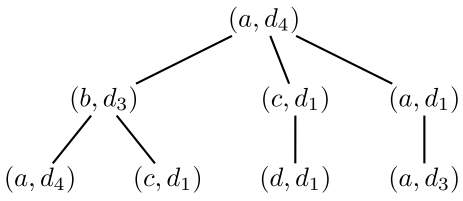
[1] Cet arbre n'est pas unique.

Un arbre étiqueté par $\Sigma \times \Sigma_{\mathcal{D}}$ peut être vu de trois manières différentes (Voir l'exemple de la Figure 6.1) :

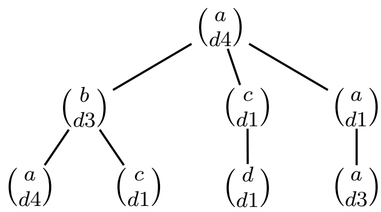
(1) un arbre au sens classique du terme étiqueté par un alphabet fini ;

(2) un arbre de données étiqueté sur Σ en considérant la partie $\Sigma_{\mathcal{D}}$ comme des valeurs de données ;

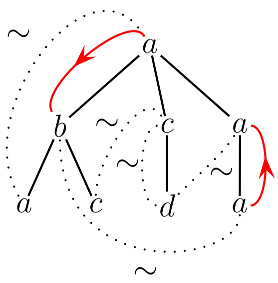
(3) un pattern d'arbre : l'arbre étiqueté sur Σ décrit la structure finie du pattern d'arbre (on considère que toutes les arêtes sont du type enfant), les étiquettes de $\Sigma_{\mathcal{D}}$ représentent les contraintes de données entre les nœuds ((in)égalité et ordre)[2].



(a) vu comme un arbre de $\Sigma \times \Sigma_{\mathcal{D}}$ (1)



(b) vu comme un arbre de données (2)



(c) vu comme un pattern d'arbre (3)

FIG. 6.1 – Un exemple d'arbre étiqueté par $\Sigma \times \Sigma_{\mathcal{D}}$.

On considère un automate fini $\mathcal{A}_L = (Q, Q_i, Q_f, \delta)$ qui reconnaît le langage L. Comme d'habitude, Q représente l'ensemble des états, Q_i (resp. Q_f) les états initiaux (resp. finaux) et δ est la fonction de transition de l'automate.

Les états de l'automate $\mathcal{A}_{\varphi,L}$ sont les éléments de $Q \times (\Sigma \times \Sigma_{\mathcal{D}})^k \times \{0,1\}^n$. Intuitivement la première partie de l'état de l'automate $\mathcal{A}$ représente un état de l'automate $\mathcal{A}_L$, la seconde partie va représenter un sous-arbre d'arité r et de profondeur d et le bit final permet de garder en mémoire si on a déjà rencontré un témoin pour chacun des patterns d'arbre de la formule.

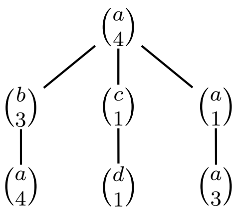
Une transition de l'automate est de la forme $(q_1, t_1, b_1), \cdots, (q_r, t_r, b_r), a \rightarrow (q, t, b)$ et vérifie :

– $q_1, \cdots, q_r, a \rightarrow q$ est une transition de $\mathcal{A}_L$;

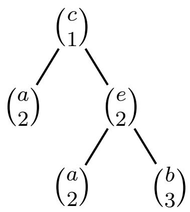
[2]Dans la figure, pour une question de lisibilité, on présente un pattern d'arbre comprenant un ensemble minimum de contraintes. Ce pattern d'arbre est équivalent au pattern d'arbre complet que l'on aurait obtenu explicitant les contraintes de manière exhaustive.

– l'arbre t vu comme un arbre de données doit satisfaire le pattern d'arbre P obtenu en regroupant sous une racine a les patterns d'arbre décrits par les arbres $t_1, \ldots, t_r$ et dont on a tronqué les feuilles pour garder une profondeur d ; (voir l'exemple de la Figure 6.2)
– le vecteur des bits b est correct en fonction des b_i et de t : un bit de b vaut 1 si il valait déjà 1 dans un des b_i ou si le pattern d'arbre correspondant apparaît dans l'arbre t.

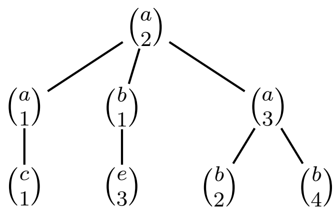
L'idée est de stocker dans un état de l'automate la représentation du sous-arbre de profondeur d de la position courante et l'information des patterns d'arbre de la formule de patterns que l'on a déjà rencontrés.



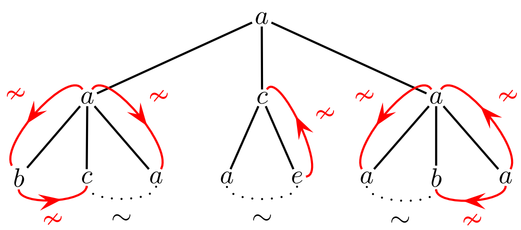
(a) L'arbre t_1



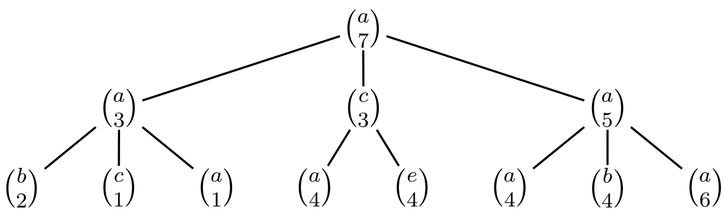
(b) L'arbre t_2



(c) L'arbre t_3



(d) Le pattern d'arbre P obtenu à partir des arbres t_1, t_2 et t_3 et de la racine a pour une profondeur $d = 3$.



(e) Un arbre t qui vérifie P.

FIG. 6.2 – Exemple des arbres t_1, t_2, t_3 et t et du pattern d'arbre P d'une transition de l'automate de la forme $(q_1, t_1, b_1)(q_2, t_2, b_2)(q_3, t_3, b_3), a \rightarrow (q, t, b)$.

Un état (q, t, b) est un état final si et seulement si q est final dans $\mathcal{A}_L$ et b est une valuation des patterns d'arbre qui satisfait la formule. Cette dernière étape se calcule en temps polynomial à partir de la formule de patterns en construisant le Parse Tree qui lui correspond.

La taille de cet automate est doublement exponentielle dans la taille de l'entrée. La construction de l'automate est aussi 2EXPTIME. La seule partie difficile est le calcul des transitions. La complexité du calcul des transitions réside dans les deux dernières conditions imposées. Ces deux étapes sont principalement des problèmes de model-checking de patterns d'arbre ou de négations de patterns d'arbre dans l'arbre t. Les patterns d'arbre ainsi que l'arbre t sont de taille exponentielle dans la taille de l'entrée. La complexité du problème de model-checking d'un pattern d'arbre est dans ce cas NP ou co-NP. Cette étape se fait donc en temps 2EXPTIME dans l'entrée.

Le problème du vide d'un tel automate est PTIME donc la construction et le vide de l'automate sont 2EXPTIME dans la taille de l'entrée.

Il reste à montrer que cet automate reconnaît bien la projection sur l'alphabet fini des modèles de la formule φ qui respectent le langage régulier. Étant donné un modèle, il est facile de voir que l'on peut retrouver une exécution acceptante de l'automate. Par définition de l'automate une exécution, dont les états contiennent le témoin du sous-arbre de profondeur d de la position courante en respectant b et les états du langage régulier, va convenir. Réciproquement étant donné un arbre sans données accepté par l'automate et une exécution acceptante. La suite des états et la définition des transitions nous assurent que l'on va pouvoir colorier cet arbre avec des valeurs de données de telle sorte qu'il vérifie bien la valuation des patterns correspondant à l'état final. Par définition des états finaux l'arbre obtenu est bien un modèle de la formule qui respecte le langage régulier. $\square$

6.1.3 Contraintes horizontales

Cette question a été évoquée dans l'introduction. Si on considère des arbres ordonnés et des patterns d'arbre qui contiennent des prédicats de structure horizontale entre ses nœuds, l'expressivité du formalisme change.

Model-Checking

Le problème de model-checking garde la même complexité. La preuve de la Proposition 4.6 s'adapte sans problème au cas des patterns contenant des contraintes structurelles horizontales. Les preuves des autres résultats gardent la même structure.

Satisfaisabilité

En ce qui concerne le problème de satisfaisabilité, il y a des différences.

Indécidabillité : De nombreux fragments deviennent indécidables si on rajoute les contraintes horizontales. La raison est la suivante : à partir du moment où on considère des arbres ordonnés, on peut coder un mot de données dans un arbre de profondeur un. La relation de successeur (resp. d'ordre) entre les positions du mot correspond alors à la relation de successeur (resp. d'ordre) horizontal entre les enfants de la racine de l'arbre. Autoriser le prédicat successeur horizontal $\triangleright$ (resp. $\triangleright^*$) dans les patterns d'arbre permet donc de coder les prédicats $+1$ (resp. $<$) des patterns de mots. D'après le Théorème 5.8, le problème de satisfaisabilité est donc indécidable pour tout fragment qui autorise la négation de patterns d'arbre ainsi que les deux prédicats $\triangleright$ et $\triangleright^*$ (y compris si on se restreint aux arbres de données de profondeur bornée).

Résultats inchangés : L'argument de pompage horizontal utilisé pour construire un modèle d'arité bornée dans certaines preuves reste valable si on n'autorise pas la relation successeur $\triangleright$. Ainsi les preuves des Théorèmes 4.27, 4.36 et 4.33 s'adaptent facilement au cas du fragment $\mathrm{BC}(\sim, |, \triangleright^*)$ (cas général ou profondeur bornée) et du fragment $\mathrm{BC}(\sim, |, \|, \triangleright^*)$ dans le cas des arbres de profondeur bornée.

Cas ouverts : Si on autorise la relation successeur $\triangleright$, l'argument de pompage horizontal n'est plus valable. De plus, dans tous les cas, l'argument de pompage vertical utilisé pour prouver l'existence d'un modèle de profondeur bornée dans la preuve de la Proposition 4.21 et du Théorème 4.24 ne fonctionne plus. Les fragments dont on ne connait pas la complexité directement sont donc $\mathrm{BC}^+(\sim, |, \|, \triangleright^*, \triangleright)$, $\mathrm{BC}(\sim, \|, \triangleright^*)$, $\mathrm{BC}(\sim, \|, \triangleright)$ et $\mathrm{BC}(\sim, |, \triangleright)$.

Pour certains fragments, on sait que la complexité va augmenter car on peut coder les mots de données horizontalement dans un arbre de profondeur un. Cela permet d'exprimer les patterns de mots. Le résultat du Théorème 5.17, nous donne la PSPACE-dureté de tout fragment qui contient des négations et le prédicat $\triangleright$. On en déduit que, si on se restreint à des arbres de profondeur bornée (borne fixée), le problème de satisfaisabilité en présence de contrainte régulière pour le fragment $\mathrm{BC}(\sim, |, \|, \triangleright)$ est PSPACE-dur alors que ce problème est Σ_2 sans ce prédicat horizontal (voir le Théorème 4.33).

6.2 Remarques sur le problème d'inclusion

Problème 6.2 Étant donné un langage régulier L et deux formule de patterns φ et ψ, le **problème d'inclusion** en présence de contrainte régulière consiste à savoir si tout arbre de donnée qui satisfait φ en respectant L satisfait aussi ψ.

On note cela $\varphi \subseteq_L \psi$.

Ce problème est très lié au problème de satisfaisabilité étudié précédemment. En effet on a $\varphi \not\subseteq_L \psi$ si et seulement si la formule de patterns $\neg\varphi \wedge \psi$ est satisfaisable en présence de L. On en déduit donc les trois premiers résultats suivants :

Corollaire 6.3 (Théorème 4.17) Le problème d'inclusion en présence de contrainte régulière est indécidable pour les fragments $\mathrm{BC}(\sim, |, \|)$, $\mathrm{BC}^-(\sim, |, \|)$ et $\mathrm{BC}^+(\sim, |, \|)$.

Corollaire 6.4 (Théorème 4.24) Le problème d'inclusion en présence de contrainte régulière est coNEXPTIME-complet pour les fragments $\mathrm{BC}(\sim, \|)$, $\mathrm{BC}^-(\sim, \|)$ et $\mathrm{BC}^+(\sim, \|)$.

Corollaire 6.5 (Théorème 4.27) Le problème d'inclusion en présence de contrainte régulière est 2EXPTIME-complet pour les fragments $\mathrm{BC}(\sim, |)$, $\mathrm{BC}^-(\sim, |)$ et $\mathrm{BC}^+(\sim, |)$.

Le résultat suivant s'obtient facilement à partir de la preuve de NP-dureté de la Proposition 4.21

Proposition 6.6 Le problème d'inclusion en présence de contrainte régulière est NP-dur pour les fragments $Ptn(\sim, |, \|)$, $Ptn(\sim, |)$ et $Ptn(\sim, \|)$.

Preuve. On utilise le codage de 3SAT décrit dans la preuve de la Proposition 4.6. Étant donnée une formule propositionnelle θ en forme 3CNF, on a un arbre de données l'arbre t_θ et un pattern d'arbre P_θ tel que le pattern d'arbre P_θ apparait dans l'arbre θ si et seulement si la formule θ est satisfaisable.

Dans la preuve de la Proposition 4.21, on considère le langage L constitué d'un seul arbre qui correspond à la projection sur l'alphabet fini de t_θ, on considère le pattern d'arbre P_{t_θ} obtenu à partir de t_θ en remplaçant les valeurs de données par les contraintes de données correspondantes. Un seul arbre vérifie P_{t_θ} et respecte L ; il s'agit de l'arbre t_θ. On en déduit le résultat suivant $P_{t_\theta} \subseteq_L P_\theta$ si et seulement si la formule θ est satisfaisable. $\square$

6.3 Comparaison avec d'autres formalismes

Dans cette partie, on explique pourquoi notre formalisme basé sur les patterns d'arbre et patterns de mots est incomparable en matière d'expressivité avec les autres formalismes dont on a parlé jusqu'à présent. Dans le cas des arbres, on s'attardera sur le formalisme XPath.

6.3.1 La logique FO^2

Dans la Partie I, on définit des logiques basées sur la logique du premier ordre à deux variables. Ces logiques sont incomparables avec notre formalisme. L'idée est que les patterns ont un pouvoir de comparaison de données très important. Cependant ils ne peuvent pas exprimer des propriétés universelles compliquées.

Voici des propriétés qui séparent les deux formalismes.

Propriété 6.7 Toute classe qui contient un a contient un b.

Cette propriété est très facile à exprimer à l'aide d'une formule de FO^2 :

$$\forall x[a(x) \rightarrow \exists y\ [y \sim x \wedge b(y)]]$$

Propriété 6.8 Il existe deux positions x et y dans la même classe telles que x est étiquetée c , y est étiquetée d, x précède y et il existe une position étiquetée par un b entre x et y.

Cette propriété s'exprime facilement à l'aide du pattern suivant : $\underbrace{c \;=\; b \;=\; d}_{\sim}$

Propriété 6.9 Il y a une bijection entre la suite des données correspondant aux positions a et la suite des données des positions b.

Cette propriété s'exprime à l'aide d'une formule de patterns simple qui est la négation des patterns suivants :

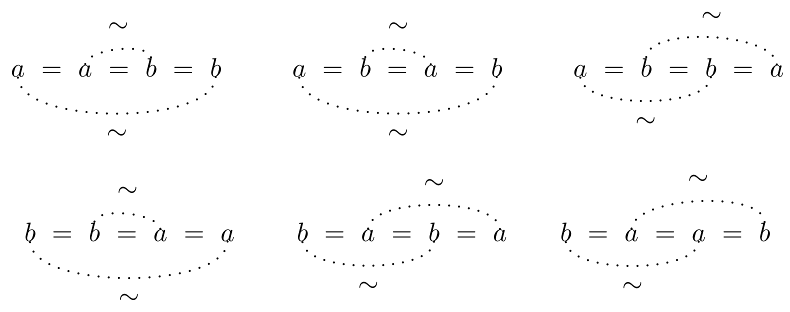


FIG. 6.3 – Les comportements interdits

La Propriété 6.7 n'est pas exprimable à l'aide d'une formule de patterns.

Si on pouvait exprimer ce type de propriété à l'aide d'une formule de patterns de BC($\sim, <$), la satisfaisabilité des formules de BC($\sim, <$) en présence de contrainte régulière serait indécidable. On considère le codage du problème de correspondance de Post utilisé dans la preuve du Théorème 2.29. Étant donnée une instance de PCP $(u_i v_i)_i$, on code une solution du problème dans un mot de données de telle sorte qu'un mot de données est une solution de l'instance de PCP considérée si et seulement si :

(1) la projection sur l'alphabet fini du mot appartient au langage $(u_i\overline{v_i})^*$

(2) toute classe contient au plus une lettre de Σ et une lettre $\bar{\Sigma}$

(3) toute classe contient qui contient une lettre de Σ contient aussi son équivalent dans $\bar{\Sigma}$ et réciproquement (ceci correspond à la propriété 6.7)

(4) la séquence de données qui correspond à la partie étiquetée Σ est la même que celle correspondant à la partie étiquetée $\bar{\Sigma}$ (cette propriété est exprimable à l'aide de patterns, cela correspond à la propriété 6.9).

Le point (1) est une propriété régulière. La propriété (2) est facilement exprimable dans BC$^-$($\sim, <$). La propriété (3) correspond à la Propriété 6.7. La propriété (4) correspond à la Propriété 6.9 que l'on sait exprimer BC$^-$($\sim, <$).

Donc si on pouvait exprimer la propriété 6.7, la satisfaisabilité des formules de BC($\sim, <$) en présence de contrainte régulière serait indécidable ce qui n'est pas le cas d'après le Théorème 4.24.

La Propriété 6.8 n'est pas exprimable dans la logique FO^2($\sim, <, +1$). (Voir le Lemme 3.31)

La Propriété 6.9 n'est pas exprimable dans la logique EMSO^2($\sim$,+1,$\oplus$1).

Cette propriété n'est pas exprimable dans la logique EMSO^2($\sim$,+1,$\oplus$1). Si c'était le cas, le problème de satisfaisabilité de cette logique serait indécidable. On reprend ce même codage de PCP utilisé dans le preuve du Théorème 2.29. La seule propriété que l'on ne sait pas exprimer dans la logique FO^2 est la propriété (4) c'est à dire la propriété 6.9. Si cette propriété était exprimable dans la logique EMSO^2($\sim$,+1,$\oplus$1), la satisfaisabilité de cette logique serait indécidable ce qui n'est pas le cas d'après la Proposition 2.91.

En terme de complexité, le problème de satisfaisabilité pour FO^2 est équivalent au problème d'accessibilité dans les réseaux de Pétri dans le cas des mots de données. Dans le cas des arbres on ne sait pas si la satisfaisabilité du fragment général est décidable. Si on autorise uniquement les prédicats de navigation successeur (c'est à dire la relation père-fils et successeur horizontal), le problème est décidable et au moins aussi dur que le problème du vide des BVASS ("Branching Vector Addition System with States"). La complexité de ce dernier problème est une question difficile ouverte depuis plusieurs années. Notre formalisme à base de patterns est donc bien meilleur en matière de complexité puisque les complexités obtenues varient entre NP et 2EXPTIME.

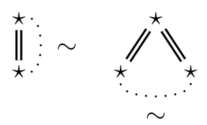
6.3.2 Les formalismes à registres

Dans la Partie I, on définit des modèles d'automates et fragments de la logique temporelle munis de registres afin de stocker et comparer des valeurs de de données. Le pouvoir d'expression de ces formalismes est aussi incomparables avec notre formalisme. Les patterns ont un pouvoir de comparaison de données très important. Cependant, comme dans le cas de la logique monadique, les automates alternants peuvent exprimer des propriétés universelles que les formules de patterns ne peuvent exprimer.

Propriété 6.10 Toutes les positions d'un arbre de données sont dans des classes différentes

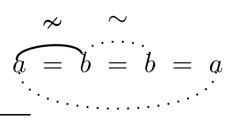
Cette propriété n'est pas exprimable par un automate $ATRA(\triangledown, \triangleright)$. L'idée est qu'à partir de déplacements vers le bas $\triangledown$ et vers la droite $\triangleright$, il n'est pas possible de construire un chemin qui passe par le fils le plus à droite de la racine et la feuille la plus à gauche de l'arbre[3]. Un automate $ATRA(\triangledown, \triangleright)$ ne semble pas pouvoir distinguer un arbre dans lequel ces deux nœuds sont dans la même classe d'un arbre dans lequels ils sont dans des classes différentes.

Cette propriété s'exprime facilement en interdisant les deux patterns d'arbre suivants :



(a) (b)

Si on se restreint au cas des mots de données, les automates ARA_1 peuvent exprimer ce type de contrainte mais ne semblent pas capable d'exprimer par exemple l'existence d'un témoin pour le pattern de mot suivant :



[3]En supposant bien sûr que la racine a au moins deux enfants.

Propriété 6.11 Pour toute position étiquetée par un a il existe une position étiquetée par un b qui est dans la même classe que le a et situé à sa droite dans le mot.

Cette propriété est facilement exprimable dans la logique temporelle par la formule suivante :

$$G(a \Rightarrow (\downarrow \wedge F(b \wedge \uparrow)))$$

Cette propriété ne peut s'exprimer à l'aide de patterns de mots. Si c'était le cas, on pourrait exprimer la Propriété 6.7. En effet, si on peut exprimer cette propriété, c'est forcément à l'aide de négations de patterns (car le fragment positif ne permet d'exprimer que des propriétés existentielles). On peut donc imaginer qu'en utilisant le même genre de patterns, on pourrait exprimer le fait que pour toute position a il existe un b dans la même classe à droite ou à gauche dans le mot de donnée. Cette propriété correspond à la Propriété 6.7 qui n'est pas exprimable à l'aide de patterns de mots comme on l'a montré précédemment.

En matière de complexité, la conclusion est la même que dans le cas de la logique à deux variables. Pour les fragments décidables des formalismes à registres présentés dans la première partie, le problème de satisfaisabilité est non primitif recursif.

6.3.3 Requêtes conjonctives :

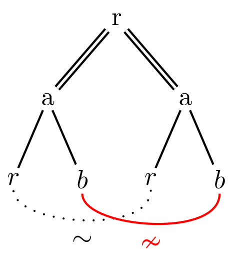
Les requêtes conjonctives peuvent s'exprimer à l'aide de patterns d'arbre. Dans [GKS04, BMS07], les auteurs considèrent des requêtes conjonctives sans contraintes de données. Dans un nouveau travail [BMS08], ils étudient ce formalisme augmenté par des contraintes régulières (schémas). Dans quelques cas, ils autorisent des comparaisons de données. En terme d'expressivité, sans prédicat horizontal de type $\triangleright^*$ ou $\triangleright$, les requêtes conjonctives sont strictement moins expressives que les combinaisons booléennes de patterns d'arbre. En effet la sémantique des requêtes est non injective et les négations ne sont pas autorisées. Étant donnée une requête conjonctive, on peut construire une formule de patterns équivalente. La taille de cette formule de patterns peut néanmoins être exponentielle dans la taille de la requête. Ce formalisme, bien que moins expressif, peut donc s'avérer plus succcinct dans certains cas. Dans [BMS08], les auteurs montrent que le problème de satisfaisabilité d'une requête est NP-complet et que le problème d'universalité est 2EXPTIME-complet. Ils montrent aussi que le problème d'universalité devient indécidable si on considère une disjonction de requêtes. Ce dernier résultat correspond au résultat d'indécidabilité du Théorème 4.17. Cependant le preuve utilisée est différente (elle utilise une réduction au problème de correspondance de Post).

6.3.4 XPath

Expressivité

Les patterns d'arbre ont un pouvoir expressif bien plus important que XPath au niveau comparaison de données car un pattern d'arbre peut comparer simultanément un nombre arbitraire de nœuds alors que XPath est limité à la comparaison de deux nœuds. La sémantique injective est aussi quelque chose qui distingue les patterns d'arbre du langage XPath. Par exemple, XPath ne peut pas exprimer le pattern d'arbre décrit dans la Figure 6.4(c). Sans considérer les contraintes de données ce pattern d'arbre n'est pas exprimable dans XPath car ce formalisme a une sémantique non injective (Xpath ne pas pas assurer que les deux nœuds a sont différents). Au delà de cette question d'injectvité, si on interprète le pattern d'arbre décrit dans la Figure 6.4(c) à l'aide d'une sémantique non-injective, il ne pourrait pas être exprimé par une requête XPath car une telle requête ne peut pas comparer simultanément plus de deux nœuds ; si on trouve deux nœuds b du bon type, il ne sera pas possible d'assurer que deux nœuds r sont bien les frères de ces nœuds b.

Cependant XPath autorise des négations internes. On peut ainsi exprimer, dans XPath, des propriétes que l'on ne peut pas exprimer à l'aide de patterns d'arbre. Un exemple est donné dans la Figure 6.4(d). Cette requête exprime le fait qu'il existe un nœud étiqueté a qui n'a pas de descendant étiqueté b dans la même classe. Intuitivement, une formule de patterns peut exprimer le fait qu'il existe un nœud a qui possède un descendant b ou bien qu'aucun nœud a n'a un descendant b, mais on ne va pas pouvoir exprimer le fait qu'il existe un nœud a qui n'a aucun descendant étiqueté b. Si on pouvait exprimer une telle propriété, on pourrait grâce à une combinaison booléenne de patterns d'arbre exprimer la Propriété 6.7, ce qui n'est pas possible.



(c) Exemple de pattern non exprimable dans XPath

$$\triangledown^* ::\mathrm{a}[\neg\ \epsilon :: \star/@A = \triangledown^* :: b/@A]$$

(d) Requête XPath non exprimable à l'aide de Pattern

Le fragment positif : Si on interdit la négation, une requête XPath peut être vue comme une disjonction de requêtes conjonctives avec des comparaisons de données. D'après la remarque sur ce type de requêtes ci-dessous, on en déduit que le fragment positif de XPath est inclus dans le fragment positif des combinaisons booléennes de patterns d'arbre. Cette inclusion reste stricte, car l'exemple donné ci-dessus est dans le fragment positif des combinaisons booléennes de patterns d'arbre.

Une extension du fragment positif : Dans [DT01], les auteurs considèrent un fragment de XPath appelé Simple-XPath. Ce fragment n'autorise que la navigation verticale mais permet des comparaisons de données à l'aide de variables. Ce type de comparaison se rapproche ainsi des contraintes de données autorisées dans les patterns d'arbre. Cependant les négations sont interdites, autant dans la partie navigationnelle que dans les comparaisons de données. Ainsi une expression du fragment Simple-XPath peut être vue comme un pattern d'arbre contenant uniquement des contraintes d'égalité de données et interprété avec une sémantique non injective. Les auteurs s'intéressent au problème d'inclusion de ces expressions en présence de schémas appelés SXIC. Ces schémas contiennent des contraintes d'intégrité comme des contraintes d'inclusion. On ne peut pas exprimer ce type de contraintes à l'aide de combinaisons booléennes de patterns d'arbre. Ce type de formalismes est donc incomparable au notre.

Evaluation

Le problème de l'évaluation de XPath a été beaucoup étudié (Voir [BK] pour un *survey* détaillé du sujet). Dans le cas de requêtes booléennes, ce problème correspond au problème de model-checking. Le problème d'évaluation de requêtes est PTime. Dans un récent article [BP08], les auteurs montrent que la complexité en donnée est même linéaire. Dans notre cas, le problème de model-checking a une complexité plus élevée puisqu'il est BH-dur dans le cas général (Théorème 4.8) et déjà NP-complet pour un seul pattern (Voir la Proposition 5.2 dans le cas des arbres et la Proposition 5.2 dans le cas des mots).

Satisfaisabilité

En ce qui concerne le problème de satisfaisabilité en présence de contraintes régulière, XPath est indécidable dans le cas général [BFG05]. Dans cet article, les auteurs considèrent la complexité du problème de satisfaisabilité de XPath en présence de DTD. Ils détaillent la complexité du problème pour de nombreux fragments de XPath en faisant varier le type de DTD considérées. Les complexités obtenues varient entre NP et NExpTime. Dans le cas de patterns, le fragment général est aussi indécidable. On obtient la décidabilité en restreignant la forme des formules, le pouvoir expressif des patterns ou encore en bornant la profondeur des modèles considérés. Les complexités ainsi obtenues varient entre NP et 2ExpTime ce qui est comparable aux résultats obtenus pour XPath. Plus précisément, la complexité du problème pour les fragments positifs de XPath (i.e. n'autorisant pas la négation) varie entre PTime et NP-complet. Pour le fragment BC($\sim, |, \|$) le problème est NP-complet et PTime si on considère un seul pattern d'arbre. En présence de négation et de comparaison de données, les auteurs de [BFG05], donnent deux résultats : le problème est ExpTime-dur pour le fragment XPath($\uparrow, \cup, [], \neg, =$) et NExpTime pour le fragment XPath($\downarrow, \cup, [], \neg, =$). Dans le cas des patterns d'arbre, le problème est 2ExpTime-complet pour le fragment BC($\sim, |$) et NExpTime-complet si on borne la taille des modèles considérés.

6.3.5 Les patterns en Bases de Données

L'idée d'utiliser des patterns afin d'exprimer des propriétés dans des bases de données semi-structurées n'est pas nouvelle. Dans ce contexte, on trouve des travaux concernant des patterns souvent sans contrainte sur les valeurs de données [WPJ03, AYCLS01, NS98]. L'objectif est en général de trouver des techniques d'optimisation pour l'évaluation de requêtes [AKJP$^+$02, BKS02, BCM04]. Dans cette thèse on ne s'intéresse pas à l'optimisation mais à trouver des bornes de complexité exactes dans le pire cas pour les problèmes de model-checking et de satisfaisabilité.

Plusieurs travaux s'intéressent à des patterns avec des contraintes de données mais une sémantique non injective. Dans [LRWZ04], les auteurs considèrent le problème de satisfaisabilité dans le cas d'un seul pattern. Dans [AMN$^+$03], les auteurs considèrent le problème de "type checking". Ce problème est plus compliqué que le problème de non-satisfaisabilité mais incomparable au problème de satisfaisabilité.

6.3.6 Data Exchange Settings

Les patterns d'arbre sont utilisés dans [AL05] afin de définir des "Data exchange settings". Un "data exchange setting" est un système de contraintes qui définit à partir d'un document XML "source" un ensemble de documents appelés "arbres cibles". L'idée à l'origine de ces travaux est la suivante : le "data exchange setting" est défini de telle manière que les informations "intéressantes" du document source sont présentes dans les arbres cibles et la structure de ces arbres rend la manipulation des informations de la source plus simple. Ce type de formalisme contient une DTD qui décrit la structure de la source, une DTD que doivent respecter les cibles et un certain nombre de contraintes liant les arbres cibles à la source. Ce type de formalisme autorise en particulier des jointures entre les valeurs de données de la source et celle des cibles à l'aide de formules contenant des contraintes de données entre les nœuds de deux patterns différents (un des patterns d'arbre correspondant à la source et l'autre à la cible).

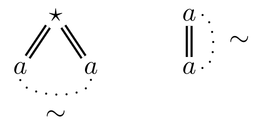
Les auteurs s'intéressent à deux problèmes : le premier est le problème de cohérence de "data exchange setting", le second est le problème de "query answering under data exchange settings".

Un "data exchange setting" est dit cohérent s'il existe un arbre qui vérifie la DTD source tel que qu'il existe au moins un arbre cible non vide. Étant données une conjonction de patterns d'arbre et une DTD, on peut construire un "data exchange setting" tel que le problème de cohérence de ce "data exchange setting" est équivalent au problème de satisfaisabilité de la conjonction de patterns d'arbre en présence de la DTD. Cependant les patterns utilisés dans ces travaux sont moins expressifs que les patterns d'arbre étudiés dans cette partie. En effet, ils ne peuvent exprimer des contraintes d'inégalité de donnée ni des combinaisons booléennes. On ne peut donc pas déduire de résultat des résultats de cohérence de ces travaux. Il serait intéressant d'étudier le problème pour des patterns plus expressifs ; en particulier autorisant les contraintes d'inégalité de données.

Le problème de "query answering" s'intéresse à l'évaluation de requêtes sur les arbres cibles plutôt que sur l'arbre source. Ce problème semble relié à notre problème de model-checking étudié. Cependant les résultats et les idées de preuve ne semblent pas s'appliquer à notre cas.

6.3.7 Contraintes d'intégrité

Des combinaisons booléennes de patterns d'arbre permettent d'exprimer des propriétés du type contraintes d'intégrité (Voir la Section 3.1). À partir du moment où on autorise la relation descendant $\|$ dans les patterns d'arbre ainsi que les négations de patterns d'arbre, on peut exprimer les "clefs". Il suffit pour cela d'interdire les deux patterns d'arbre présentés dans la Figure 6.4. Les combinaisons booléennes de patterns d'arbre ne permettent pas d'exprimer les contraintes de type "inclusion". En effet la Propriété 6.7 est une propriété de type contrainte d'inclusion qui n'est pas exprimable à l'aide de patterns d'arbre (Voir la partie 6.3.1). La propriété de type "clef étrangère" étant la combinaison d'une contrainte de "clef" et d'une contrainte d'inclusion, on ne peut pas non plus l'exprimer à l'aide de patterns d'arbre.



(e) (f)

FIG. 6.4 – Les patterns d'arbre à interdire pour assurer que a soit une clef.

6.4 Problèmes ouverts

Model-Checking : Il serait intéressant de connaître la complexité du problème de model-checking si on fixe la formule de patterns. Ce problème n'est pas évident même dans le cas d'un seul pattern d'arbre.

Extension - contraintes horizontales : La question de la décidabilité et de la complexité du problème de satisfaisabilité pour certains fragments de cette extension restent ouverts. On sait en particulier que dans certains cas, la complexité augmente. Les fragments pour lesquels le problème reste ouvert sont : $\mathrm{BC}^+(\sim, |, \|, \triangleright^*, \triangleright)$, $\mathrm{BC}(\sim, \|, \triangleright^*)$, $\mathrm{BC}(\sim, \|, \triangleright)$ et $\mathrm{BC}(\sim, |, \triangleright)$.

Problème d'inclusion : Dans le cas des fragments $Ptn(\sim, |, \|)$, $Ptn(\sim, \|)$, $Ptn(\sim, |)$, on connait un résultat de NP-dureté mais on ne connaît pas les complexité exactes.

Data Exchange : Comme on l'a expliqué dans la partie 6.3.6, il serait intéressant de voir comment se comporte ce type de formalisme en considérant des patterns d'arbre plus expressifs, en particulier en présence de contraintes d'inégalité de données.

Extension - langage de requête : Dans ce travail, on considère uniquement des formules booléennes. On pourrait imaginer étendre notre formalisme de pattern d'arbre à l'aide de variables afin de définir un langage de requête. La façon d'étendre le formalisme est déjà une question non-triviale. La première question est de savoir ce que vont extraire les requêtes. La solution classique est d'extraire des nœuds de l'arbre. Dans ce cas les variables introduites dans la formule de patterns correspondent à des nœuds de l'arbre. Selon les règles de syntaxe autorisées, les variables vont permettre d'effectuer des jointures entre les nœuds de l'arbre. Une autre solution est d'extraire des valeurs de données. Dans ce cas les variables vont correspondre à des valeurs de données et pourront éventuellement permettre des jointures entre les valeurs de données des différents patterns d'arbre de la formule de patterns. Dans les deux cas, la gestion de la négation des patterns d'arbre n'est pas évidente non plus. Une question importante est de savoir comment gérer sémantiquement le cas de jointures entre un pattern d'arbre positif et la négation d'un pattern d'arbre. Les formalismes classiques considèrent en général des conjonctions de requêtes où les négations (quand elles sont autorisées) n'affectent pas directement les nœuds variables.

Chapitre 7

Conclusion

Logiques et automates pour les mots de données

Le premier travail de cette thèse porte sur l'extension de la logique du premier ordre à l'aide d'un prédicat binaire $\sim$ permettant de comparer les valeurs de données des positions d'un mot de données. Le problème de satisfaisabilité est indécidable en général (Théorème 2.29) mais on montre que le fragment restreint aux formules utilisant deux variables $\mathrm{FO}^2(\sim, < +1)$ est décidable. Ce résultat utilise une correspondance entre logique et automates. En effet, on définit une classe d'automates appelés automates à données et on montre que le langage décrit par une formule du fragment à deux variables peut être reconnu par un tel automate. Le vide de ces automates est décidable. En terme d'expressivité la classe des automates à données est équivalente au fragment de la logique existentielle du second ordre $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$.

De nombreuses extensions de la logique $\mathrm{FO}^2(\sim, <)$ proposées mènent à l'indécidabilité : trois variables, deux relations d'équivalence, relation d'ordre sur les données. Le problème de satisfaisabilité reste décidable si on ajoute les "successeurs étendus", le "successeur de classe", des quantificateurs existentiels du second ordre ou si on considère le modèle étendu des mots de données infinis.

Du point de vue de la complexité, le vide des automates, ainsi que le problème de satisfaisabilité du fragment à deux variables est équivalent au problème du vide des automates multicompteurs aussi connus sous le nom de VASS ("Vector Addition System with States"). Ce problème est décidable mais sa complexité exacte est une question ouverte depuis longtemps. La meilleure borne inférieure est ExpSpace et l'on ne connaît pas de borne supérieure élémentaire. Il serait intéressant d'approfondir cette étude afin de mieux comprendre la difficulté en complexité de ces deux modèles. En particulier, la réduction de la décidabilité des formules vers le problème du vide des automates multicompteurs est 2ExpTime. On ne sait pas si cette réduction est optimale.

Si on restreint l'expressivité logique, on obtient des bornes de complexité élémentaires. Le Théorème 2.59 montre que le problème de satisfaisabililté est NExpTime-complet pour la logique $\mathrm{FO}^2(\sim, <)$. Dans le cas de la logique $\mathrm{FO}^2(\sim, +1)$, le Théorème 2.63 montre que le problème est 3NExpTime. La meilleure borne inférieure connue pour ce problème est NExpTime, il serait intéressant de connaître la complexité exacte du problème.

Logiques et automates pour les arbres de données

La logique FO^2, s'étend au cas des arbres de données. On montre que le problème de satisfaisabilité est décidable pour le fragment $FO^2(\sim, +1)$. La preuve nous donne une borne de complexité 3NExpTime pour ce problème. La complexité exacte du problème n'est pas conuue, la meilleure borne inférieure étant NExpTime-difficile. La décidabilité du problème de satisfaisabilité pour le fragment $FO^2(\sim, <, +1)$ reste un problème ouvert. Ce problème est au moins aussi difficile que le problème du vide des TVASS, une extension aux arbres des automates multicompteurs.

On s'intéresse au lien de ces résultats avec la problématique de description de schémas pour les documents XML. Un document XML pouvant être vu comme un arbre de données, le résultat de décidabilité entraîne la décidabilité de tout problème d'analyse statique exprimable comme un problème de satisfaisabilité d'une formule de la logique $EMSO^2(\sim, +1)$. On pense en particulier au problème de cohérence de schéma : la logique $EMSO^2(\sim, +1)$ permet de combiner n'importe quel langage régulier d'arbre avec des propriétés de clé et clé étrangère unaire. De la même manière, on obtient la décidabilité du problème d'inclusion en présence de schéma pour toutes contraintes et schémas exprimables dans $FO^2(\sim, +1)$. Ce résultat étend le résultat donné dans [AFL05, FL02] qui était restreint aux clés et clés étrangères et aux schémas de type DTD.

Un autre résultat intéressant découle de la comparaison de ces formalismes avec le langage XPath. Le problème de satisfaisabilité en présence de schéma est décidable pour tout fragment XPath et schéma exprimables par une formule $FO^2(\sim, +1)$. On identifie, en particulier, un fragment de XPath permettant des comparaisons de données restreintes. Dans le fragment $LocalDataXPath$ la navigation correspond aux axes locaux mais est augmentée par un nouvel axe $\boxtimes$ qui permet de se déplacer dans une position quelconque du document. On impose des restrictions syntaxiques sur les tests de données afin de pouvoir réduire la satisfaisabilité et le problème d'implication pour les expressions $LocalDataXPath$ au problème de satisfaisabilité de la logique $FO^2(\sim, +1)$. On obtient ainsi un nouveau fragment décidable de XPath avec comparaisons de données.

Il est intéressant de remarquer que les formalismes décrits permettent de regrouper les requêtes XPath et les schémas et contraintes d'intégrité. Ces formalismes étaient jusqu'ici étudiés en général séparément.

On regrette cependant la complexité élevée des problèmes considérés. Le problème de satisfaisabilité de la logique $FO^2(\sim, +1)$ est 3NExpTime et -NExpTime-dur et le problème pour le fragment plus général $FO^2(\sim, <, +1)$, s'il est décidable aura une complexité au moins ExpSpace et probablement plus importante.

Combinaisons booléennes de patterns

On propose un formalisme basé sur des combinaisons booléennes de "patterns ". Les patterns sont des objets simples qui permettent d'exprimer des propriétés sur des arbres ou des mots de données. Un pattern peut à la fois exprimer des contraintes sur la structure finie du modèle mais aussi des contraintes d'(in)égalité sur les données. En terme d'expressivité ce formalisme est incomparable aux formalismes précédemment décrits. Un pattern ou plus généralement une combinaison booléenne de patterns permettent des comparaisons de données plus générales que les formalimes logiques mais ne permettent pas de récursion ni de "négations imbriquées".

Le problème de satisfaisabilité est indécidable en général ; on propose deux façons d'obtenir la décidabilité du problème. La première consiste à restreindre les formules considérées ou l'expressivité des patterns. La seconde est de borner la profondeur des modèles considérés, ce qui est naturel dans le contexte du langage XML. Un ensemble de documents est ainsi en général de profondeur bornée et faible mais d'arité quelconque. Dans les deux cas, on obtient alors des résultats de décidabilité avec des complexités allant de NP-complet à 2ExpTime-complet.

Le problème du model checking est toujours décidable. On montre qu'il est déjà NP-complet si l'on ne considère qu'un seul pattern d'arbre et que ce problème est $P^{\|NP}$ et BH-dur dans le cas général.

Si l'on regarde le cas restreint des mots de données on obtient des résultats similaires. La principale différence est la complexité du problème de satisfaibilité ; la structure linéaire des mots par rapport à la structure branchante des arbres permet d'économiser une exponentielle.

On remarque aussi que l'extension à des modèles avec plusieurs relations d'équivalence ou un ensemble de données munies d'une relation d'ordre préservent les résultats de décidabilité ce qui n'est pas le cas dans la première partie.

Le tableau suivant récapitule l'ensemble des résultats obtenus.

Fragments	Model-Checking	Satisfaisabilité	*bnd* Sat	bnd_f Sat
BC($\sim, \|, \mid$)	$P^{\|NP}$ et BH-dur	Indécidable	NExpTime-complet	Σ_2-complet
BC($\sim, \mid$) Mots	$P^{\|NP}$ et BH-dur PTime	2ExpTime-complet PSpace-complet	NExpTime-complet	Σ_2-complet
BC($\sim, \|$) Mots	$P^{\|NP}$ et BH-dur $P^{\|NP}$ et BH-dur	NExpTime-complet Σ_2-complet	NExpTime-complet	Σ_2-complet
BC^-($\sim, \|, \mid$) Mots	coNP-complet coNP-complet	Indécidable Indécidable	NExpTime-complet	Σ_2-complet
BC^+($\sim, \|, \mid$) Mots	NP-complet NP-complet	NP-complet NP-complet	NP-complet	NP-complet

Conformément à notre objectif, les résultats de complexité sont plus raisonnables que dans le cas de la logique du premier ordre. Ces résultats de complexité ne permettent cependant pas d'envisager l'utilisation en pratique de ces formalismes mais enrichissent les résultats théoriques dans le domaine.

Les patterns d'arbre ont un pouvoir expressif bien plus important que XPath en termes de comparaison de données car un pattern d'arbre peut comparer simultanément un nombre arbitraire de nœuds alors que XPath est limité à la comparaison de deux nœuds. Cependant XPath autorise des négations internes ce qui n'est pas autorisé dans le cas des patterns. Si on interdit la négation, une requête XPath peut être vue comme une disjonction de requêtes conjonctives avec des comparaisons de données. Le fragment positif de XPath est strictement inclus dans le fragment positif des combinaisons booléennes de patterns d'arbre.

En terme de complexité, le problème de l'évaluation des requêtes booléennes de XPath correspond au problème de model-checking. La complexité de ce problème est PTIME dans le cas de XPath (et même linéaire si on considère la complexité en données). Dans notre cas, le problème de model-checking a une complexité plus élevée puisqu'il est $P^{\|NP}$ et BH-dur dans le cas général et déjà NP-complet pour un seul pattern dans le cas des mots.
En ce qui concerne le problème de satisfaisabilité en présence de contrainte régulière, XPath est indécidable dans le cas général [BFG05]. La complexité obtenue varie entre NP et NEXPTIME lorsque l'on considère des fragments de XPath et en faisant varier le type de contrainte régulières considérées. Dans le cas de patterns, le fragment général est aussi indécidable. On obtient la décidabilité en restreignant la forme des formules, le pouvoir expressif des patterns ou encore en bornant la profondeur des modèles considérés. Les complexités ainsi obtenues varient entre NP et 2EXPTIME ce qui est comparable aux résultats obtenus pour XPath.

Il serait intéressant d'étendre les patterns afin de pouvoir exprimer des contraintes sur la structure horizontale du document, même si le problème de satisfaisabilité est indécidable dans les cas les plus généraux.

On pourrait aussi envisager d'étendre le formalisme des patterns d'arbre à l'aide de variables pour définir un langage de requête. Cela entraîne un certain nombre de questions comme le choix de la nature des variables (nœuds ou valeurs de données) ou encore la question de la sémantique de la négation dans ce cadre.

Bibliographie

[ABB80] Jean-Michel Autebert, Joffroy Beauquier, and Luc Boasson. Langages sur des alphabets infinis. *Discrete Applied Mathematics*, 2 :1–20, 1980.

[AD90] Rajeev Alur and David L. Dill. Automata For Modeling Real-Time Systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming, ICALP'90*, volume 443, pages 322–335. Springer-Verlag, 1990.

[AD93] Rajeev Alur, , and David L. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104(1) :2–34, 1993.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[ADR07] Parosh A. Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized Verification of Infinite-State Processes with Global Conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157. Springer-Verlag, 2007.

[AFL05] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. Consistency of XML Specifications. In *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 15–41. Springer-Verlag, 2005.

[AH93] Rajeev Alur and Thomas A. Henzinger. Real-Time Logics : Complexity and Expressiveness. *Information and Computation*, 104(1) :35–77, 1993.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins : A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering, ICDE'02*, pages 141–153. IEEE Computer Society, 2002.

[AL05] Marcelo Arenas and Leonid Libkin. XML data exchange : consistency and query answering. In *Proceedings of the 24th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'05*, pages 13–24. ACM Press, 2005.

[AMN+03] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values : typechecking revisited. *Journal of Computer and System Sciences*, 66(4) :688–727, 2003.

[ASV06] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and querying XML with incomplete information. *ACM Transactions on Database Systems*, 31(1) :208–254, 2006.

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. *SIGMOD Records*, 30(2) :497–508, 2001.

[BB07a] Henrik Björklund and Mikolaj Bojanczyk. Bounded Depth Data Trees. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming, ICALP'07*, volume 4596 of *Lecture Notes in Computer Science*, pages 862–874. Springer-Verlag, 2007.

[BB07b] Henrik Björklund and Mikolaj Bojanczyk. Shuffle Expressions and Words with Nested Data. In *Proceedings of the 32nd Mathematical Foundations of Computer Science International Symposium, MFCS'07*, volume 4708 of *Lecture Notes in Computer Science*, pages 750–761. Springer-Verlag, 2007.

[BC07] Nicole Bidoit and Dario Colazzo. Testing XML constraint satisfiability. *Electr. Notes Theor. Comput. Sci.*, 174(6) :45–61, 2007.

[BCM04] Véronique Benzaken, Giuseppe Castagna, and Cédric Miachon. CQL : a Pattern-based Query Language for XML. In *Actes des 20èmes Journées Bases de Données Avancées,BDA'04*, pages 469–490, 2004.

[BCT04] Nicole Bidoit, Serenella Cerrito, and Virginie Thion. A First Step towards Modeling Semistructured Data in Hybrid Multimodal Logic. *Journal of Applied Non-Classical Logics*, 14(4) :447–475, 2004.

[BDM+06] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *Proceedings of the 25th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'06*, pages 10–19, 2006.

[BEH95] Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. On the Verification Problem of Nonregular Properties for Nonregular Processes. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, LICS'95*, pages 123–133. IEEE Computer Society, 1995.

[Bès08] Alexis Bès. An Application of the Feferman-Vaught Theorem to Automata and Logics for Words over an Infinite Alphabet. *CoRR*, abs/0801.2498, 2008.

[BFG05] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath Satisfiability in the Presence of DTDs. In *Proceedings of the 24th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'05*, pages 25–36. ACM Press, 2005.

[BH96] Ahmed Bouajjani and Peter Habermehl. Constrained Properties, Semilinear Systems, and Petri Nets. In *Proceedings of the 7th International Conference on Concurrency Theory, CONCUR'96*, volume 1119, pages 481–497. Springer-Verlag, 1996.

[BHJS07] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting Systems with Data. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory, FCT'07*, volume 4639 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2007.

[BJS07] Ahmed Bouajjani, Yan Jurski, and Mihaela Sighireanu. A Generic Framework for Reasoning About Dynamic Networks of Infinite-State Processes. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 690–705. Springer-Verlag, 2007.

[BK] Michael Benedikt and Christoph Koch. XPath Leashed. To appear in ACM Computing Surveys.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins : Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD'02*, pages 310–321. ACM Press, 2002.

[BMS+06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-Variable Logic on Words with Data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, LICS'06*, pages 7–16. IEEE Computer Society, 2006.

[BMS07] Henrik Björklund, Wim Martens, and Thomas Schwentick. Conjunctive Query Containment over Trees. In *Proceedings of the 11th Database Programming Languages07*, volume 4797 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2007.

[BMS08] Henrik Björklund, Wim Martens, and Thomas Schwentick. Optimizing Conjunctive Queries over Trees using Schema Information. In *Proceedings of the 33rd Mathematical Foundations of Computer Science International Symposium, MFCS'08*, volume 5162 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[Boj08] Mikolaj Bojanczyk. Tree-Walking Automata. In *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications, LATA'08*, volume 5196 of *Lecture Notes in Computer Science*, pages 1–2. Springer-Verlag, 2008.

[BP08] Mikolaj Bojanczyk and Pawel Parys. XPath evaluation in linear time. In *Proceedings of the 27th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'08*, pages 241–250. ACM Press, 2008.

[BPT03] Patricia Bouyer, Antoine Petit, and Denis Thérien. An Algebraic Approach to Data Languages and Timed Languages. *Information and Computation*, 182(2) :137–162, 2003.

[BS07] Henrik Björklund and Thomas Schwentick. On Notions of Regularity for Data Languages. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory, FCT'07*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2007.

[CDG+07] Hubert Comon, M. Dauchet, R. Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, and Sophie Tison and. Tree Automata Techniques and Applications, 2007. release October, 12th 2007.

[CG08] Christian Choffrut and Serge Grigorieff. Finite n-tape automata over possibly infinite alphabets : Extending a theorem of Eilenberg et al. . *Theoretical Computer Science*, 2008.

[CGH+88] Jin-Yi Cai, Thomas Gundermann, Juris Hartmanis, Lane A. Hemachandra, Vivian Sewelson, Klaus W. Wagner, and Gerd Wechsung. The Boolean hierarchy I : structural properties. *SIAM J. Comput.*, 17(6) :1232–1252, 1988.

[CH86] Jin-Yi Cai and Lane A. Hemachandra. The Boolean hierarchy : hardware over NP. In *Proceedings of the Conference on Structure in Complexity Theory*, pages 105–124. Springer-Verlag, 1986.

[CK96] Richard Chang and Jim Kadin. The Boolean Hierarchy and the Polynomial Hierarchy : A Closer Connection. *SIAM J. Comput.*, 25(2) :340–354, 1996.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1), 1981.

[Dav04] Claire David. Mots et Données Infinies. Master Thesis, Université Paris 7, LIAFA, 2004.

[Dav08] Claire David. Complexity of Data Tree Patterns over XML Documents. In *Proceedings of the 33rd Mathematical Foundations of Computer Science International Symposium, MFCS'08*, volume 5162 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[DG08] Stéphane Demri and Régis Gascon. Verification of Qualitative Z-constraints. *Theoretical Computer Science*, 2008. To appear.

[Dic13] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35 :413–422, 1913.

[DL06] Stéphane Demri and Ranko Lazić. LTL with the Freeze Quantifier and Register Automata. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, LICS'06*, pages 17–26. IEEE Computer Society, 2006.

[DL08a] Stéphane Demri and Ranko Lazić. LTL with the Freeze Quantifier and Register Automata. To appear inACM Transactions on Computational Logic, 2008.

[DL08b] Stéphane Demri and Denis Lugiez. Complexity of Modal Logics with Presburger Constraints. Research Report LSV-08-25, Laboratoire Spécification et Vérification, ENS Cachan, France, 2008.

[DLN07] Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL : Decidability and complexity. *Information and Computation*, 205(1) :2–24, 2007.

[DLS08] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking freeze LTL over one-counter automata. In *Proceedings of the 11th Foundations of Software Science and Computational Structures, FoSSaCS'08*, volume 4962 of *Lecture Notes in Computer Science*, pages 490–504. Springer-Verlag, 2008.

[DT01] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases, KRDB01*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[EMSS92] E. Allen Emerson, Aloysius K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative Temporal Reasoning. *Real-Time Systems*, 4(4) :331–352, 1992.

[EVW02] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-Order Logic with Two Variables and Unary Temporal Logic. *Information and Computation*, 179(2) :279–295, 2002.

[FL02] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3) :368–406, 2002.

[Für83] Martin Fürer. The computational complexity of the unconstrained limited domino problem (with implications for logical decision problems). In *Logic and Machines : Decision Problems and Complexity*, volume 171 of *Lecture Notes in Computer Science*, pages 312–319. Springer-Verlag, 1983.

[GF05] Floris Geerts and Wenfei Fan. Satisfiability of XPath Queries with Sibling Axes. In *Proceedings of the 10th Database Programming Languages05*, volume 3774 of *Lecture Notes in Computer Science*, pages 122–137. Springer-Verlag, 2005.

[GGS04] Philippe de Groote, Bruno Guillaume, and Sylvain Salvati. Vector Addition Tree Automata. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, LICS'04*, pages 64–73. IEEE Computer Society, 2004.

[Gis81] Jay L. Gischer. Shuffle Languages, Petri Nets, and Context-Sensitive Grammars. *Communications of the ACM*, 24(9) :597–605, 1981.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB'02*, pages 95–106. Morgan Kaufmann, 2002.

[GKS04] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive Queries over Trees. In *Proceedings of the 23rd SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'04*, pages 189–200, 2004.

[GO99] Erich Grädel and Martin Otto. On Logics with Two Variables. *Theoretical Computer Science*, 224(1-2) :73–113, 1999.

[HP79] John E. Hopcroft and Jean-Jacques Pansiot. On the Reachability Problem for 5-Dimensional Vector Addition Systems. *Theoretical Computer Science*, 8 :135–159, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[JL07] Marcin Jurdziński and Ranko Lazić. Alternation-free modal mu-calculus for data trees. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, LICS'07*, pages 131–140. IEEE Computer Society, 2007.

[JL08] Marcin Jurdziński and Ranko Lazić. Alternating Automata on Data Trees and XPath Satisfiability. 2008.

[KF94] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theoretical Computer Science*, 134(2) :329–363, 1994.

[KO05] Emanuel Kieronski and Martin Otto. Small Substructures and Decidability Issues for First-Order Logic with Two Variables. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS'05*, pages 448–457. IEEE Computer Society, 2005.

[Kos82] S. Rao Kosaraju. Decidability of Reachability in Vector Addition Systems. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, STOC'82*, pages 267–281. ACM Press, 1982.

[Koy90] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4) :255–299, 1990.

[Laz08] Ranko Lazić. Safety alternating automata on data words. *CoRR*, abs/0802.4237, 2008.

[Lib06] Leonid Libkin. Logics for Unranked Trees : An Overview. *CoRR*, abs/cs/0606062, 2006.

[Lip76] J.R. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

[LLW95] François Laroussinie, Kim Larsen, and Carsten Weise. From Timed Automata to Logic - and Back. In *Proceedings of the 20th Mathematical Foundations of Computer Science International Symposium, MFCS'95*, volume 969 of *Lecture Notes in Computer Science*, pages 529–539. Springer-Verlag, 1995.

[LRWZ04] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng (Jessica) Zhao. On Testing Satisfiability of Tree Pattern Queries. In *Proceedings of the 30th International Conference on Very Large Data Bases, VLDB'04*, pages 120–131, 2004.

[Lyn97] James F. Lynch. Pebble Games in Model Theory. In *Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht*, pages 66–83. Springer-Verlag, 1997.

[Mar05] Maarten Marx. First Order Paths in Ordered Trees. In *Proceedings of the 10th International Conference on Database Theory, ICDT'05*, volume 3363 of *Lecture Notes in Computer Science*, pages 114–128. Springer-Verlag, 2005.

[May84] Ernst W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM J. Comput.*, 13(3) :441–460, 1984.

[Min67] Marvin Minsky. *Computation : finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[NS98] Andreas Neumann and Helmut Seidl. Locating Matches of Tree Patterns in Forests. In *Proceedings of the 18th Foundations of Software Technology and Theoretical Computer Science, FSTTCS'98*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145. Springer-Verlag, 1998.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3) :403–435, 2004.

[Sch07] Thomas Schwentick. Automata for XML - A survey. *Journal of Computer and System Sciences*, 73(3) :289–315, 2007.

[SI00] Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theoretical Computer Science*, 231(2) :297–308, 2000.

[SSMH04] Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. Counting in Trees for Free. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming, ICALP'04*, volume 3142 of *Lecture Notes in Computer Science*, pages 1136–1149. Springer-Verlag, 2004.

[Sto74] Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, MIT, Cambridge, Massasuchets, USA, 1974.

[W3C] W3C. Overview over SGML Ressources. available at http ://www.w3.org/Markup/SGML.

[W3C99a] W3C. XML Path Language (XPath), W3C Recommendation. available at http ://www.w3.org/TR/xpath, 1999.

[W3C99b] W3C. XSL Transformations (XSLT), W3C Recommendation. available at http ://www.w3.org/TR/xslt, 1999.

[W3C06] W3C. Extensible Markup Language (XML), W3C Recommendation. available at http ://www.w3.org/TR/xml, 2006.

[W3C07] W3C. XQuery, W3C Recommendation. available at http ://www.w3.org/TR/xquery, 2007.

[wag88] On Restricting the Access to an NP-Oracle. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming, ICALP'88*, pages 682–696. Springer-Verlag, 1988.

[WI07] Philipp Weis and Neil Immerman. Structure Theorem and Strict Alternation Hierarchy for FO^2 on Words. In *Proceedings of the 21st EACSL Annual Conference on Computer Science and Logic, CSL'07*, volume 4646 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, 2007.

[Wil94] Thomas Wilke. Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata. In *Proceedings of the 3rd Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'94*, volume 863 of *Lecture Notes in Computer Science*, pages 694–715. Springer-Verlag, 1994.

[WPJ03] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the 19th International Conference on Data Engineering, ICDE'03*, pages 443–454. IEEE Computer Society, 2003.