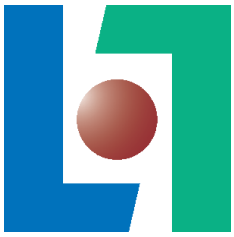


Mémoire de Recherche
Février 2008 - Juin 2008

Séparation d'espaces d'états représentés par des langages
Application au calcul d'interpolant sur les ROBDD

Nicolas Aucouturier
Université de Bordeaux 1
Master 2 Sciences et Technologie, spécialité Informatique
parcours Ingénierie des Systèmes Critiques

Encadrement :
Emmanuel Fleury LaBRI - CNRS UMR 5800
Frédéric Herbreteau LaBRI - CNRS UMR 5800

Résumé

Dans le cadre de l'approche CEGAR du model-checking, on cherche à calculer un interpolant entre deux ROBDD. Cependant, la définition des interpolants, nous permet facilement de décider si un ROBDD est un interpolant de deux autres, mais on cherche une méthode efficace pour calculer un interpolant entre deux BDD. On ne se restreindra pas au calcul d'interpolants sur les BDD. On verra en effet que les BDD peuvent être vus comme des langages, et on orientra donc nos travaux sur la recherche d'un interpolant entre deux langages.

Dans ce rapport on verra comment à partir de la notion de préfixe discriminant, on peut créer des interpolants sur les langages. En utilisant les résultats obtenus sur les langages, on regardera comment construire un interpolant de manière efficace sur des automates reconnaissant un langage de Σ^n, à structures en DAG. Enfin, on constatera que ces résultats sont transposables aux ROBDD. On étudira donc comment construire à partir de la notion de préfixes discriminants, des interpolants sur les ROBDD à l'aide d'algorithmes efficaces.

On analysera ensuite les résultats obtenus dans le calcul d'interpolants, et particulièrement d'un interpolant ayant une représentation d'une taille plus faible que les interpolés. Ces résultats seront obtenus par l'implémentation des algorithmes grâce à la libraire BuDDy.

Remerciments

Je tiens à remercier les personnes qui m'ont amené vers les techniques de vérification formelle, et donc en premier lieu, le responsable de mon Master (Ingénierie des Systèmes Critiques), M. Griffault. Il avait été présent pour me recevoir lors du choix, difficile, d'une spécialité de Master 2.

Je voudrais aussi remercier MM. Zeitoun et Desbarats, qui m'ont dirigé vers un Master Recherche, sans jamais me forcer la main. M. Zeitoun a été très attentif à mes moindres questions sur le déroulement de ce Master Recherche et sur ces débouchés.

Il me faut aussi grandement remercier MM. Fleury et Herbreteau, mes encadrants de mémoire. Ils ont, à ma demande, proposé un sujet de mémoire. Ils ont toujours été là pour répondre à mes questions, et me pousser à regarder plus loin. Je voudrais les remercier particulièrement, pour les réponses en moins d'un quart d'heure, même le dimanche !

Je tiens aussi à saluer les gens qui m'ont soutenus pendant toute la durée de ce mémoire, et qui ont été présent jusqu'à la fin pour relire et corriger.

Table des matières

Chapitre 1

Introduction

1.1 Motivation de la vérification

La société actuelle exploite l'informatique dans de plus en plus de domaines, tant par l'utilisation de micro-ordinateurs personnels, que pour les systèmes industriels, ou encore dans les systèmes embarqués. Cette dépendance permanente, nous oblige à définir, pour l'informatique, des critères de qualité, et de robustesse, et ce dans le but de qualifier les différents systèmes utilisant l'informatique.
On donne souvent, aux systèmes ayant une nécessité forte de robustesse, le nom de *systèmes critiques*. Cette catégorie regroupe des systèmes de différents domaines, tels que le transport de personnes, le nucléaire, la santé, l'industrie de masse. Leur point commun est surtout le fait qu'une défaillance au sein d'un système de ce type va avoir un coût important, en termes de vies humaines, d'intérêts financiers, ou encore d'un point de vue écologique. Quelque soit la qualité du programmeur et les conditions pour lesquelles il développe un système, les erreurs (bogue, oubli de code, omission d'un comportement, ...) sont inévitables. Il est donc nécessaire de passer par une étape de vérification. Pour s'en persuader, il existe quelques exemples (ou catastrophes) célèbres.

Le premier exemple que l'on peut citer, sûrement parce qu'il m'est plus contemporain que d'autres, est le crash du vol 88 [1], vol inaugural, d'Ariane V (4 juin 1996), utilisant la fusée Ariane 501. Cet accident a beaucoup terni l'image d'Ariane Espace, et ne fut pas sans conséquence financière. Brièvement et sans rentrer dans les détails, ce crash a été causé par la défaillance d'un système mis au point pour Ariane IV, système complètement fonctionnel sur cette dernière, ayant été réutilisé sans les précautions nécessaires pour Ariane V. En effet, dans ce système, la représentation de la poussée des moteurs d'Ariane IV était codée par un entier signé de 16 bits (valeurs de -32768 à 32767), mais la représentation de cette même poussée pour les moteurs d'Ariane V dépassait de cette plage de valeurs, ce qui provoqua une trajectoire folle de la fusée menant à son autodestruction après 39 secondes de vol.

Un autre exemple célèbre est le dysfonctionnement des centraux téléphoniques d'AT&T, en 1990. C'est une mise à jour du système d'exploitation de ces commutateurs téléphoniques qui est à l'origine de la défaillance. Elle a causé de nombreuses pertes d'appels, et le blackout de tout le réseau téléphonique de la côte Est des Etats-Unis pendant 9 heures (le 15 janvier 1990). L'erreur de cette mise à jour paraissait pourtant minime. Le programmeur ayant fait cette mise à jour à introduit une instruction `switch` en langage C. La syntaxe en C du `switch` permet d'exécuter les cas suivant le cas en cours, si on le souhaite. Mais si on ne le souhaite pas, il faut rajouter l'instruction `break`. Cependant le programmeur en a oublié un. Cet oubli a provoqué l'exécution des instructions d'un cas et l'exécution des instructions du cas suivant (en l'occurrence le cas par

1. Le rapport de l'accident produit par le comité d'enquête indépendant est disponible sur http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

défaut). Cet oubli, puisqu'il s'agit d'un simple oubli de code, a donc coûté très cher à AT&T, alors qu'une campagne de vérification aurait sûrement permis d'éviter cela.

Ces deux exemples, bien qu'ayant un coût financier important, n'ont pas mis en danger la population de façon directe. Un exemple dramatique fut l'utilisation par les centres de soins contre le cancer du Therac-25, de 1985 à 1987, au Canada. En effet, dans ce cas des erreurs logicielles mirent deux ans à être décelées et admises. Ces erreurs provoquaient en cas de saisie trop rapide des paramètres d'irradiation, la délivrance de doses aléatoires. Certains des cas dans lesquels les doses étaient extrêmement fortes (facteur 100 des doses prescrites) furent "reconnues" comme erreur du Therac-25 , mais d'autres erreurs de dosages n'ont peut être pas été reconnues (dosages trop faibles, dosages peu différents, ...). Ces erreurs furent dramatiques pour les personnes qui se retrouvant sur-irradiées sont décédées, mais aussi sûrement pour les personnes sous-irradiées qui n'ont pas été soignées.

Ces différents exemples, bien que non exhaustifs pour les défaillances de systèmes critiques, montrent bien la nécessité de vérifier les systèmes, et surtout les systèmes les plus critiques.

1.2 Techniques de vérification

Nous sommes maintenant persuadés de la nécessité de la vérification, la question qui se pose désormais est de savoir comment faire cette vérification. L'action de vérifier consiste en fait à s'assurer qu'un système satisfait une ou des propriétés données. Il existe au moins deux grandes approches de la vérification. Une famille regroupe les différents types de *tests*, et une autre famille regroupe les *méthodes formelles*. Les tests sont bien connus des programmeurs, qui les utilisent en général pour tous leurs programmes. Cependant, le contenu de ce rapport sera axé sur les méthodes formelles, qui ont l'avantage par rapport aux tests d'être exhaustives sur les états du système.

Pour continuer à diviser les différentes techniques de vérification, attaquons nous maintenant aux différents types de vérification par les méthodes formelles. On trouve ici, de manière non exhaustive, trois grandes classes : l'analyse statique, la preuve de théorèmes (et preuve de modèles), et le model-checking (vérification de modèles).

- L'**analyse statique** consiste à vérifier l'absence de bogues usuels, et cela sans exécuter le système. On recherche en fait à calculer algorithmiquement des propriétés sur le comportement du système. Cette analyse, bien que relativement efficace dans certaines conditions, est fortement restreinte, en particulier par l'application directe du Théorème de Rice.
- La **preuve de modèles**, pour sa part, consiste en la conception d'un modèle vérifiant des propriétés mathématiques fixées. Le modèle peut être créé soit automatiquement à partir d'un code source, soit par un opérateur humain (dans certains cas, on peut dériver du modèle le code source correspondant). Les propriétés à prouver sont quant à elles des théorèmes que l'on va essayer de prouver avec les hypothèses du modèle, les preuves sont en partie automatisées par l'assistant de preuve. Cependant, cet outil n'est qu'un assistant nécessitant l'aide de l'homme. Cette nécessité de l'aide de l'homme est induite par le théorème d'incomplétude de Gödel. En effet, selon un corollaire à ce théorème, il est acquis qu'on ne peut pas créer un algorithme qui prouve tous les théorèmes de l'arithmétique. Or l'assistant de preuve pourrait avoir besoin de prouver une partie de l'arithmétique que son algorithme ne peut prouver.
- La dernière classe est celle qui nous intéresse. En effet, ce rapport porte sur le **model-checking**. Le principe du model-checking est de s'assurer que le modèle, représentant un système, vérifie une propriété donnée. Pour cela le model-checking explore les différents comportements du modèle. Pour parcourir ces différents comportements, le model-checker explore de façon systématique l'espace des états du système. Cependant, le model-checking présente des inconvénients, le premier vient de la construction du modèle, qui doit fidèlement représenter les comportements du système. L'autre désagrément vient de l'explosion de l'espace d'états du modèle, qui empêche le calcul exhaustif, par une machine, des successeurs

ou des prédécesseurs d'un état donné. On verra dans la suite qu'il existe des théories pour résoudre ce problème.

Chapitre 2

Model-Checking et approche CEGAR

2.1 Généralité

Le model-cheking est donc un type de vérification qui cherche à prouver qu'un système, et plus particulièrement que le modèle d'un système, satisfait une propriété que le concepteur considère comme importante. Pour cela, la technique consiste à explorer tous les états accessibles du système.

Pour cela, le système à analyser (le terme système correspond aussi bien à un programme, qu'à un algorithme, ou à un protocole, ...) doit être modélisé en une structure de graphe nommée structure de Kripke.

Définition 1. *Une structure de Kripke est un 4-uplets $\mathcal{K} = (Q, I, E, L)$ tel que*
- *Q, un ensemble fini d'états,*
- *$I \subseteq Q$, l'ensemble des états initiaux,*
- *$R \subseteq Q^2$, une relation de transition,*
- *$L : Q \to 2^{AP}$, fonction d'étiquetage des états de Q par un ensemble de propositions atomiques de AP.*

Ensuite, la propriété à vérifier doit être écrite sous la forme d'une formule de logique, sur les formules atomiques de la structure de Kripke. Il existe deux grandes classes de logiques pour représenter les propriétés que l'on souhaite vérifier. Les logiques temporelles linéaires représentent l'ensemble des comportements (LTL, ...). Et les logiques arborescentes pour écrire des propriétés sur l'arbre des comportements (CTL, CTL*, TCTL, ...). Il existe plusieurs types de propriétés, par exemple :

Vivacité : Le système ne rencontre jamais d'état bloquant,

Accessibilité : Le système peut atteindre un état vérifiant une propriété donnée,

Sûreté : Le système ne peut pas atteindre d'état vérifiant une propriété donnée,

...

Avec un système modélisé par une structure de Kripke, et une propriété écrite dans une logique particulière, le travail du model-checker est alors de combiner le modèle et la propriété pour vérifier si le modèle vérifie ou non cette propriété. Si le modèle ne vérifie pas la propriété, le model-checker fournira un contre-exemple permettant de justifier sa réponse. Ce contre-exemple sera une suite d'états, de la structure de Kripke, du modèle, qui fait que le système ne vérifie pas la propriété. Un contre-exemple est un chemin dans la structure de Kripke du modèle.

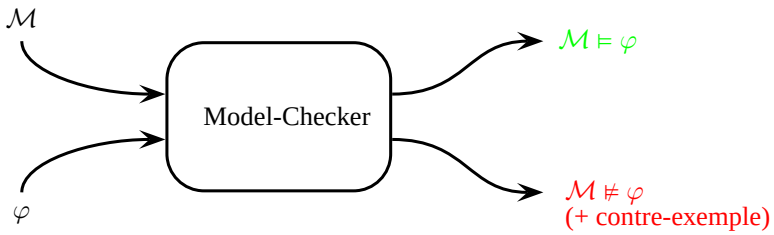


FIGURE 2.1 – Model-Checking du modèle $\mathcal{M}$ pour la propriété φ

Le principe d'un model-checker est habituellement représenté par le schéma de la figure 2.1. Et on note usuellement le fait qu'un modèle, $\mathcal{M}$, vérifie (ou satisfasse) une propriété, φ, par $\mathcal{M} \vDash \varphi$.

Dans le cadre de ce rapport, ce qui va nous intéresser sera plus particulièrement le model-checking de formule de LTL. L'algorithme (expliqué dans [1]) de model-checking des systèmes, représentés par une structure de Kripke, pour les formules de LTL, consiste au marquage successif des états du système en fonction de ses caractéristiques (vérifie ou non une proposition atomique), et cela en suivant les indications données par la propriété que l'on souhaite vérifier.

Théorème 1 ([1]). *Le model-checking d'une structure de Kripke, $\mathcal{M}$, vis-à-vis d'une propriété de LTL, φ, a une complexité en temps de $\mathcal{O}(|\mathcal{M}||\varphi|)$.*

L'avantage du model-checking est que le travail est automatique et sûr, et qu'il analyse d'une manière exhaustive les comportements du système. Donc une propriété validée pour un modèle est à coup sûr vraie sur le modèle (et sur le système, si le modèle le représente bien). Cet avantage d'exhaustivité (par rapport aux tests) apporte aussi des défauts. L'explosion combinatoire du nombre d'états est problématique. En effet si on imagine que le nombre d'états d'un modèle est en fait, le produit du nombre d'états de tous ses sous-systèmes ; et qu'il faut un sous-système par variable, qui a lui-même un état par valeur de la variable ; on arrive vite à des tailles de modèles énormes. Le model-checking devient alors exponentiel en la taille de l'entrée.
Pour répondre à cette problématique, a été introduite la notion de model-checking symbolique [2]. Dans ce type de model-checking, la structure de Kripke, du modèle, est représentée sous la forme d'une fonction booléenne. Cette représentation booléenne sert autant à représenter la fonction de transition du modèle que les différents ensembles d'états. Pour représenter ces différentes fonctions booléennes, la représentation sous forme de BDD (Binary Decision Diagram), expliquée par Bryant dans [3], se révèle très efficace.
L'utilisation de la représentation symbolique n'étant pas suffisante, pour de gros systèmes, l'abstraction de modèles fut introduite [4]. Elle consiste au partitionnement, judicieux, de l'ensemble des états du modèle, et au calcul de la nouvelle fonction de transition entre ces parties, pour pouvoir effectuer un model-checking. Le modèle calculé est appelé modèle abstrait, et noté $\mathcal{M}^{abs}$. Cependant cette phase d'abstraction doit, par définition, conserver les comportements du modèle concret, mais contient alors des comportements non présents dans le modèle concret (modèle d'origine). On dit alors que le système abstrait simule le système concret. Cela nous permet de dire que le modèle vérifie une propriété si son modèle abstrait vérifie aussi cette propriété. Mais ne nous permet pas de dire si un modèle ne satisfait pas une propriété, si son modèle abstrait ne vérifie pas cette propriété. Dans ce cas, le contre-exemple produit par le model-checker doit être présent (simulable) dans le modèle concret pour que la propriété soit insatisfaite dans le modèle concret. Cependant si ce contre-exemple n'existe pas dans le modèle concret, on ne peut rien dire sur notre modèle vis à vis de cette propriété, et ce pour l'abstraction courante. En effet, un partitionnement de l'ensemble d'états du système concret peut ne pas nous fournir suffisamment d'information pour décider si une propriété est satisfaite ou non ; alors qu'un autre partitionnement créera, éventuellement, un autre système abstrait nous permettant de conclure vis-à-vis de cette propriété.

A cause de ces problèmes avec l'abstraction, a été mis en place [5] un cycle d'abstraction et

de raffinement se basant sur le contre-exemple produit par le model-checker. Ce cycle est appelé cycle CEGAR (Counter-Example Guided Abstraction Refinement).

2.2 L'approche CEGAR

Le principe de ce cycle d'abstraction raffinement guidé par le contre-exemple (CEGAR) [5] est d'appliquer l'algorithme de model-checking, non pas directement sur le modèle concret, mais sur des abstractions de ce dernier. L'algorithme peut nécessiter plusieurs passes de model-checking, mais sur des abstractions différentes du modèle. De plus, on se limitera pour l'approche CEGAR à des propriété de sûreté. Cette notion d'abstraction consiste à la création d'un modèle, $\mathcal{M}^{abs}$, qui doit simuler le modèle concret, $\mathcal{M}$. La simulation de $\mathcal{M}'$ par $\mathcal{M}$, notée $\mathcal{M} \preceq \mathcal{M}'$, indique que tous les comportements de $\mathcal{M}$ sont dans $\mathcal{M}'$. Cette relation de simulation implique, sur les propriétés, φ, en LTL (une logique moins expressive que ACTL*), que si $\mathcal{M}' \vDash \varphi$ alors $\mathcal{M} \vDash \varphi$. Ce qui nous permet de dire que si un modèle abstrait vérifie une propriété alors son modèle concret la vérifie aussi. Par contre la réciproque n'est pas vraie, on ne peut rien dire si $\mathcal{M}' \nvDash \varphi$.

2.2.1 Le cycle CEGAR

Le cycle CEGAR est souvent représenté par le schéma de la figure 2.2. Cette figure a l'avantage de bien refléter la notion de cycle, et de montrer les principales étapes. La propriété que l'on souhaite vérifier, est une propriété de sûreté, c'est à dire on cherche à vérifier qu'on ne peut pas atteindre un état d'erreur.

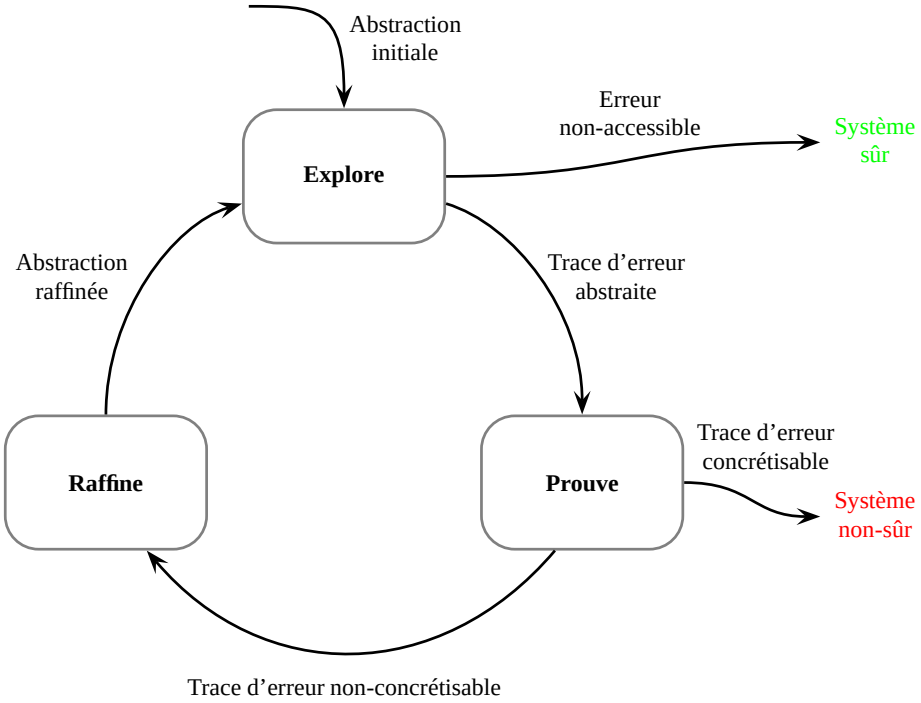


FIGURE 2.2 – Cycle CEGAR

L'étape *Explore* est l'utilisation du model-checking sur le modèle abstrait du model-checking. Comme on l'a vu précédemment, on ne peut pas tirer les mêmes conclusions que pour un model-checking simple. Mais on peut cependant, si le modèle abstrait vérifie la non-accessibilité de l'état "Erreur", affirmer que le modèle concret n'atteindra pas cet état d'erreur. Par contre, si la propriété

n'est pas vérifiée (i.e. si on accède à un état d'erreur) il nous faut vérifier la conformité du contre-exemple fourni avec le modèle concret. Cette nouvelle étape est nommée sur la figure *Prouve*. La figure 2.3 montre la concrétisation d'un contre-exemple abstrait. Les états abstraits étant représentés par des ronds et leurs concrétisations dans des rectangles. On observe qu'il est possible d'aller de la concrétisation de l'état $\hat{1}$ à l'état concret 9, mais il est de là impossible d'aller dans la concrétisation de l'état d'erreur, $\hat{4}$. La transition abstraite entre les états $\hat{3}$ et $\hat{4}$ étant donnée par la transition concrète entre 7 et 12. Notre contre-exemple abstrait n'est donc pas valide dans le modèle concret. Dans l'étape *Prouve*, il est possible, si le contre-exemple se concrétise dans le modèle concret, de dire que le modèle ne vérifie pas la propriété. Sinon, on ne peut rien dire, et il nous faut continuer le cycle. On dit alors que le contre-exemple est fallacieux ("spurious counterexample"). Pour prouver qu'un contre-exemple est fallacieux, on concrétise le contre-exemple. Cela revient à calculer pour chaque état abstrait l'ensemble des états appartenant à sa concrétisation, qui sont accessibles depuis l'état précédent du contre-exemple. On calcule alors les S_i qui représentent les états, accessibles à partir de S_{i-1}, concrétisant le $i^{\text{ème}}$ état du contre-exemple. S_1 contenant les états initiaux du modèle concret appartenant à la concrétisation du premier état du contre-exemple. Et s'il existe un $S_i = \varnothing$, alors le contre-exemple est fallacieux. Le $(i-1)^{\text{ème}}$ état d'un contre-exemple fallacieux, $S_{i-1} \neq \varnothing$, est nommé état de faute ("failure state").

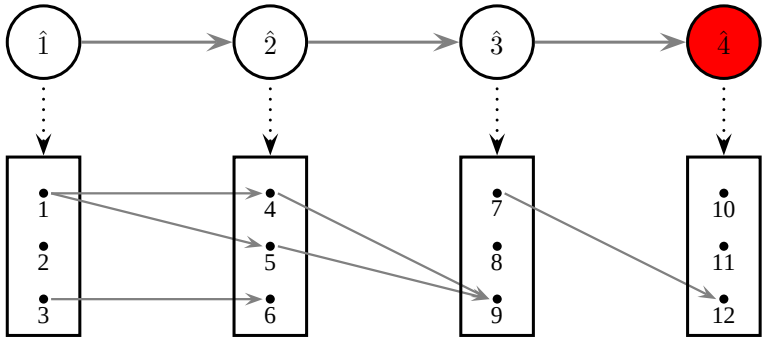


FIGURE 2.3 – Un contre-exemple abstrait et sa concrétisation [5]

On peut séparer les états concrets, appartenant à la concrétisation de cet état de faute, en trois catégories.

- Les états puits ("dead-end state"), qui sont accessibles, mais qui ne permettent pas d'atteindre d'autres états, notés S_D.
- Les états néfastes ("bad state"), qui sont inaccessibles, mais permettent d'atteindre d'autres états, notés S_B.
- Les autres états ("irrelevant state"), qui ne sont ni puits, ni néfastes, notés S_I.

Sur la figure 2.3, l'état $\hat{3}$ est un état de faute, et la décomposition de sa concrétisation donne $S_D = \{9\}$, $S_B = \{7\}$ et $S_I = \{8\}$.

Grâce à cet état de faute, et à cette partition de la concrétisation de ce dernier, on va pouvoir faire l'étape de raffinement, *Raffine*. Le raffinement va consister à créer un nouveau modèle abstrait, tel que les états de S_B et les états de S_D ne correspondent pas au même état abstrait. Cependant, il a été démontré que le fait de trouver le meilleur raffinement est NP-hard [5]. On va donc voir qu'il existe plusieurs méthodes pour faire le raffinement. Une fois un raffinement trouvé, on recommence le cycle comme avec la première abstraction.

Revenons au problème du choix d'un raffinement, il existe plusieurs heuristiques de calcul. Une solution donnée dans [5] est de prendre d'un côté les états de S_D et d'un autre les états de S_B et de S_I. On peut remarquer que le calcul d'un raffinement va revenir au calcul d'un interpolant entre S_D et S_B.

2.2.2 Raffinement : calcul d'un interpolant

On cherche à trouver un nouveau partitionnement de l'ensemble des états du système concret. On fixe cependant des impératifs, aucun état de S_D ne doit se retrouver avec un état de S_B. Notre nouveau partitionnement ne sera en plus pas complètement différent du partitionnement précédent. En effet, on parle vraiment de raffinement de l'abstraction, on garde donc le partitionnement précédent, que l'on repartitionne. Si on prend le cas particulier, qui nous intéresse, de l'état de faute, on souhaite diviser cet état en deux de telle sorte que S_D soit dans une partie et S_B dans une autre.

Définition 2 (Interpolant)**.**
Soient D et B deux ensembles tels que $D \cap B = \varnothing$ alors l'ensemble I est un interpolant de (D, B) si et seulement si,
- $D \subseteq I$, *et*
- $B \cap I = \varnothing$.

Les deux interpolants les plus faciles à calculer sont D et $\overline{B}$, il est évident que ces deux ensembles sont bien des interpolants de (D, B).

On sait que $S_D \cap S_B = \varnothing$ et on cherche un I tel que $I \subseteq S_D$ et $I \cap S_B = \varnothing$. Ce que l'on cherche est donc bien un interpolant, comme défini dans la définition 2. Cependant, on ne se place pas dans le cadre de la représentation des ensembles d'états sous forme de ROBDD, donc sous forme de formules de logique. Il existe alors une définition 3 d'un interpolant particulier, aux formules de logique, l'interpolant de Craig.

Définition 3 (Interpolant de Craig [6])**.**
Soient D et B une paire de formules avec $D \wedge B$ insatisfiable alors la formule I est un interpolant de Craig de (D, B) si et seulement si,
- $D \Longrightarrow I$,
- $B \wedge I$ *est insatisfiable, et*
- I *n'utilise que les variables communes à D et à B.*

Craig [6] a montré que pour les formules de logique du premier ordre, il existe toujours un interpolant de Craig. On est donc sûr que pour les deux ROBDD représentant S_D et S_B, il existe un interpolant de Craig.

On a vu qu'il existe plusieurs types d'interpolants, et surtout deux interpolants triviaux. La nécessité de calculer un interpolant est alors nécessaire pour réduire la taille en mémoire de la représentation de S_D ou de $\overline{S_B}$. Dans ce but, on définira une relation d'ordre permettant de comparer la taille de la représentation de deux ensembles d'états. Et on cherchera donc un algorithme ayant une efficacité raisonnable, produisant un interpolant de taille inférieur aux interpolés.

Chapitre 3

Interpolation sur les langages

Dans ce chapitre, on cherche à définir un interpolant entre deux langages, notés par analogie avec le chapitre précédent $\mathcal{L}_D$ et $\mathcal{L}_B$. Un interpolant, $\mathcal{L}_I$, entre deux langages disjoints, $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$, se doit de contenir le premier, $\mathcal{L}_D \subseteq \mathcal{L}_I$, et d'être disjoint du deuxième, $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$. Cette définition nous permet juste de dire si un langage est interpolant de deux autres. L'inconvénient est que l'on cherche une méthode pour construire un interpolant. On va donc voir, dans cette partie, comment construire un interpolant à partir de ce que l'on appellera des préfixes discriminants.

3.1 Rappels sur les langages

Un *alphabet*, Σ, est un ensemble fini de symboles. Un *mot*, sur l'alphabet Σ, est une séquence finie de symboles de Σ. Et un *langage* est un ensemble de mots. On note $|m|$, la longueur du mot m, c'est à dire le nombre de symboles de m, et m_i, le $i^{\text{ème}}$ symbole de m, $1 \leq i \leq |m|$.
On note Σ^n l'ensemble des mots sur Σ de longueur exactement n, et Σ^* l'ensemble des mots fini sur Σ, de longueur quelconque. On note la concaténation des mots $u, v \in \Sigma^*$, $u.v$. Enfin, on définit sur les mots, les notions de préfixe et de suffixe.

Définition 4 (préfixe, suffixe)**.** *Soient $u, w \in \Sigma^*$.*
- *On dit que w est préfixe de u, si $\exists v \in \Sigma^*$ tel que $w.v = u$.*
- *On dit que w est suffixe de u, si $\exists v \in \Sigma^*$ tel que $v.w = u$.*

On définit aussi la relation d'ordre préfixe sur les mots.

Définition 5 (ordre préfixe)**.** *Soit $u, w \in \Sigma^*$, alors*

$$w|u \iff \exists v \in \Sigma^*,\ w.v = u.$$

3.2 Préfixes discriminants sur les langages quelconques

Dans le but de construire notre interpolant, on cherche à connaître les différences entre deux langages donnés. Pour cela, on définit la notion de préfixe discriminant. Cette définition nécessite de savoir si un langage comporte des mots ayant un préfixe donné.

Définition 6 (préfixe d'un langage)**.**
Soit $\mathcal{L}_D \subseteq \Sigma^*$, $w \in \Sigma^*$ est un préfixe de $\mathcal{L}_D$ si et seulement si

$$\exists u \in \Sigma^* \text{ tel que } w.u \in \mathcal{L}_D.$$

Par analogie avec un préfixe d'un mot, on note un préfixe d'un langage $w|\mathcal{L}_D$.

Avec cette définition du préfixe d'un langage, on peut se donner une intuition sur les préfixes discriminants. On souhaite en effet, trouver les différences entre deux langages. Donc un préfixe discriminant est un préfixe d'un langage, qui n'est pas préfixe de l'autre.

Regardons sur un exemple ce que peut donner cette notion de préfixe discriminant. Cet exemple nous permettra aussi de voir que cette notion n'est pas symétrique.

Exemple 1. *Prenons $\mathcal{L}_D,\ \mathcal{L}_B \subseteq \Sigma^4$ deux langages sur l'alphabet $\Sigma = \{a, b\}$. Avec $\mathcal{L}_D = \{aaab\}$ et $\mathcal{L}_B = \{aabb\}$. Alors*

- *a et aa sont des préfixes de $\mathcal{L}_D$ non discriminants pour $\mathcal{L}_B$ car a et aa sont des préfixes de $aaab \in \mathcal{L}_D$ et de $aabb \in \mathcal{L}_B$.*
- *aaa est un préfixe de $\mathcal{L}_D$ discriminant pour $\mathcal{L}_B$ car aaa est préfixe de $aaab \in \mathcal{L}_D$ et il n'existe pas de mot de $\mathcal{L}_B$ pour lequel aaa soit préfixe.*
- *a et aa sont des préfixes de $\mathcal{L}_B$ non discriminants pour $\mathcal{L}_D$ car a et aa sont des préfixes de $aabb \in \mathcal{L}_B$ et de $aaab \in \mathcal{L}_D$.*
- *aab est un préfixe de $\mathcal{L}_B$ discriminant pour $\mathcal{L}_B$ car aab est préfixe de $aabb \in \mathcal{L}_B$ et il n'existe pas de mot de $\mathcal{L}_D$ pour lequel aab soit préfixe.*

Maintenant que l'exemple précédent nous a donné une idée sur les préfixes discriminants, définissons-les formellement.

Définition 7 (préfixe discriminant)**.**
Soit $\mathcal{L}_B \subseteq \Sigma^*$, $w \in \Sigma^*$ est un préfixe discriminant pour $\mathcal{L}_B$, noté $w|_d\mathcal{L}_B$, si et seulement si

$$\forall u' \in \Sigma^*,\ w.u' \notin \mathcal{L}_B$$

Dans l'exemple précédent et dans la définition 7, on voit que la notion de préfixe discriminant n'est pas symétrique. Cette définition est écrite dans le cas du calcul d'un interpolant, et la notion d'interpolant et elle-même non symétrique. En effet, on cherche à augmenter la taille d'un des deux langages, sans prendre de mots du deuxième.

On note l'ensemble des préfixes, du langage $\mathcal{L}_D$, discriminants pour le langage $\mathcal{L}_B$, $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$. En prenant l'ordre préfixe sur les mots de Σ^*, on peut définir l'ensemble des éléments minimaux de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, noté $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$. On a donc $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}$. On a alors :

$$\begin{aligned}\Omega_{\mathcal{L}_D,\mathcal{L}_B} &= \{w \in \Sigma^* \mid (w|\mathcal{L}) \text{ et } (w|_d\mathcal{L}_B)\}\\ \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} &= \{w \in \Omega_{\mathcal{L}_D,\mathcal{L}_B} \mid \nexists w' \in \Omega_{\mathcal{L}_D,\mathcal{L}_B}, (w'|w)\}\end{aligned}$$

On peut se demander si cet ensemble, $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, est non vide et unique. On prouvera dans la suite que cet ensemble est non-vide, sous certaines conditions, $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ et $\mathcal{L}_D \neq \varnothing$. Son unicité est quant à elle, induite par le fait que chaque mot de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ admette un unique plus petit préfixe dans $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$.
Pour pouvoir définir notre interpolant, on verra dans la suite, que l'on peut utiliser $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, comme ensemble de préfixes de notre interpolant. Mais pour qu'il y ait un intérêt à utiliser cet ensemble il faut qu'il soit calculable.

Théorème 2.

Si $\mathcal{L}_D$ et $\mathcal{L}_B$ sont des langages réguliers, alors $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est un langage régulier calculable.

Démonstration. Pour prouver que $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est un langage régulier calculable, on va faire une preuve constructive du fait qu'il soit régulier. On aurra alors aussi prouver la calculabilité. Pour prouver qu'il est régulier, on commencera par prouver que $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ est régulier, puis on définit alors facilement $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$.

- On pose $\mathcal{L}_{D\ pref}$, le langage qui contient $\mathcal{L}_D$ et tous les préfixes de $\mathcal{L}_D$. $\mathcal{L}_D$ étant régulier, $\mathcal{L}_{D\ pref}$ est aussi régulier [1].

1. On construit $\mathcal{L}_{D\ pref}$, par la modification de l'automate reconnaissant $\mathcal{L}_D$, sur cet automate les états co-accessibles deviennent acceptants. Il existe donc un automate décrivant ce langage, en conséquence ce langage est régulier.

– $\Omega_{\mathcal{L}_D,\mathcal{L}_B} = \Big(\text{préfixes de } \big(\mathcal{L}_{D\ pref}.\Sigma^* \cap \overline{\mathcal{L}_B}\big)\Big) \cap \mathcal{L}_{D\ pref}$ car les préfixes discriminants qui composent $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ sont les préfixes de $\mathcal{L}_D$ qui ne sont pas des préfixes de $\mathcal{L}_B$. Par cette construction, il est clair que $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ est régulier.
– $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} = \Omega_{\mathcal{L}_D,\mathcal{L}_B} \setminus \big((\Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^+) \cap \Omega_{\mathcal{L}_D,\mathcal{L}_B}\big)$, en effet les éléments de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ sont les éléments de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ qui ne restent pas dans $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ quant on leurs rajoute, au moins, un symbole. Par cette définition de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, on a évidemment un langage régulier.

On a donc bien $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ un langage régulier. En suivant la preuve, on arrive, à partir de $\mathcal{L}_D$ et de $\mathcal{L}_B$, à construire $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, donc $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est calculable. □

3.3 Préfixes discriminants sur les langages disjoints

Dans l'optique de calculer notre interpolant, on doit faire la restriction que $\mathcal{L}_D$ et $\mathcal{L}_B$ sont disjoints, pour respecter la définition d'un interpolant entre $\mathcal{L}_D$ et $\mathcal{L}_B$. On se restreint aussi aux langages de Σ^n, on verra dans la suite, sur un exemple, qu'on ne peut pas généraliser nos résutats à Σ^*. Dans notre contexte, cette restriction ne pose pas de problème, puisqu'on travaillera après sur des ROBDD, qui ont des langages de mots de longueur bornée fixée.
On souhaite savoir si $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est non vide, toujours pour le calcul de notre interpolant. Comme les deux langages sont disjoints, chaque mot de $\mathcal{L}_D$ est donc forcément différent de tous les mots de $\mathcal{L}_B$, tous les mots de $\mathcal{L}_D$ sont donc discriminants, et appartiennent donc à $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$.

Lemme 1.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$. Si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors $\mathcal{L}_D \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}$

La preuve de ce lemme est triviale, en prenant l'intuition fournie avant sa formulation. De plus, il nous permet de définir un corollaire, qui va être très intéressant pour la définition d'un interpolant. Ce lemme n'est vrai que grâce à la restriction des langages à Σ^n. En effet, pour s'en convaincre, on peut prendre le contre-exemple suivant. Soient $\Sigma = \{a\}$, $\mathcal{L}_D = \{a\}$, et $\mathcal{L}_B = \{aa\} \subseteq \Sigma^*$, a n'est pas discriminant pour $(\mathcal{L}_D, \mathcal{L}_B)$, et aa n'est pas discriminant pour $(\mathcal{L}_D, \mathcal{L}_B)$. $(\mathcal{L}_D, \mathcal{L}_B)$ n'admet donc pas de préfixe discriminant, $\Omega_{\mathcal{L}_D,\mathcal{L}_B} = \varnothing$. En généralisant, si on a deux langages $\mathcal{L}_D, \mathcal{L}_B \in \Sigma^*$, disjoints, et tels que $\mathcal{L}_B$ contienne des mots qui ont un préfixe qui est un mot de $\mathcal{L}_D$, alors $\Omega_{\mathcal{L}_D,\mathcal{L}_B} = \varnothing$. Comme nous cherchons à obtenir un interpolant à partir des préfixes discriminants, nous souhaitons qu'il existe des préfixes discriminants, d'où la restriction aux langages de Σ^n. En fait, on aurait pu faire une restriction plus faible que l'appartenance à Σ^n. En effet, comme le montre la généralisation précédente, ce que l'on souhaite c'est que $\exists w \in \mathcal{L}_D$ tel que $\forall w' \in \mathcal{L}_B$, $|w| \geq |w'|$. Mais, dans notre contexte d'interpolant sur les ROBBD (voir chapitre 5), la restriciton aux langages de Σ^n est plus appropriée. Il est aussi évident que si $\mathcal{L}_D = \varnothing$ alors $\mathcal{L}_D$ n'admet pas de préfixe discriminant, et donc $\Omega_{\mathcal{L}_D,\mathcal{L}_B} = \varnothing$ et $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} = \varnothing$. On va donc aussi supposer que $\mathcal{L}_D$ soit non vide.

Corollaire 1.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$. Si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors $\Omega_{\mathcal{L}_D,\mathcal{L}_B} \neq \varnothing$.

On sait donc que $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ est non vide si les langages sont disjoints, et qu'ils contiennent des mots de taille fixée. Cependant les résultats les plus intéressants pour notre interpolant sont ceux sur $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, pour des raisons de taille de l'interpolant résultant, et de facilité de calcul. On va donc chercher comme sur $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, si $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est non vide.

Lemme 2.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$. Si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors :

$$\forall u \in \mathcal{L}_D,\ \exists w \in \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} \text{ tel que } w|u.$$

Démonstration.
Par définition, on a $w \in \Omega_{\mathcal{L}_D,\mathcal{L}_B} \iff \exists v \in \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ tel que $v|w$. Grâce à cette équivalence, on applique le lemme 1. Ce qui nous donne : si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors $\forall u \in \mathcal{L}_D$, $\exists w \in \Omega_{\mathcal{L}_D,\mathcal{L}_B}$ tel

que ($w|u$ et $\exists v \in \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ tel que $v|w$). Ce qui nous permet de dire que si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors $\forall u \in \mathcal{L}_D,\ \exists v \in \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ tel que $v|w|u$ donc $v|u$. □

Grâce à ce lemme, on peut arriver à cette proposition importante dans le calcul de notre interpolant, on verra dans la section suivante en quoi ce résultat est intéressant.

Proposition 1.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$. Si $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$ alors $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} \neq \varnothing$.

La démonstration se fait par application directe du lemme 2.

3.4 Langage interpolant

Le but de ce travail sur les langages est de trouver un interpolant [6, 7] entre $\mathcal{L}_D$ et $\mathcal{L}_B$ (définition 2). En d'autre termes, on cherche à calculer un langage qui contienne $\mathcal{L}_D$ sans jamais contenir d'éléments de $\mathcal{L}_B$. Mais cela dans la situation particulière où les langages sont disjoints. On cherche donc un $\mathcal{L}_I$ tel que $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$. On a défini dans la section précédente un moyen de calculer les préfixes d'un langage qui n'appartiennent pas à l'autre langage, $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ et $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$. On va alors voir dans cette partie, comment à partir de cet ensemble de préfixes on va pouvoir définir un interpolant. Pour cela, on peut voir que si des préfixes permettent de différencier les langages, on peut rajouter n'importe quels éléments derrière ces préfixes pour avoir des langages toujours disjoints.

3.4.1 Interpolant par préfixes discriminants

Si on prend l'ensemble $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, tous les mots qui commencent par un mot de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ ne sont pas dans $\mathcal{L}_B$, mais peuvent être dans $\mathcal{L}_D$.

Lemme 3.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$, tels que $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$, alors :

$$\mathcal{L}_D \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \text{ et } \Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing.$$

Démonstration.

- ($\mathcal{L}_D \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^*$)
  En appliquant le lemme 1, on a $\mathcal{L}_D \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}$. Donc on a $\mathcal{L}_D \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^*$.
- ($\Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing$)
  En utilisant la définition 7 des préfixes discriminants et la définition de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, on a $\forall w \in \Omega_{\mathcal{L}_D,\mathcal{L}_B},\ \forall x',\ w.x' \notin \mathcal{L}_B$. On n'a donc aucun mot commençant par w dans $\mathcal{L}_B$. On a donc $\Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing$.

□

Dans le but de calculer un autre langage interpolant, et surtout en cherchant à agrandir sa taille, on n'utilise plus $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, mais $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$. La taille du langage sera plus grande, il contiendra plus de mots, car les préfixes étant plus petits cela laisse plus de possibilités pour compléter les mots. L'utilisation de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ est plus efficace dans le cas où on représente les langages par des ROBDD (chapitre 5).

Théorème 3.
Soient $n \geq 0$ et $\mathcal{L}_D, \mathcal{L}_B \subseteq \Sigma^n$, tels que $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$, alors :

$$\mathcal{L}_D \subseteq \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \text{ et } \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing.$$

Démonstration.

– $(\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing)$
En utilisant la deuxième partie du lemme 3, et le fait que $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B} \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}$ et donc, par construction $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \subseteq \Omega_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^*$. On a $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \cap \mathcal{L}_B = \varnothing$.
– $(\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \supseteq \mathcal{L}_D)$
Pour l'autre partie, de manière similaire à la preuve du théorème 3, mais en utilisant le lemme 2, on obtient $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}.\Sigma^* \supseteq \mathcal{L}_D$.

□

On a donc construit deux interpolants de $(\mathcal{L}_D, \mathcal{L}_B)$. On notera le second $\mathcal{L}_I = \{w.x | w \in \widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ et $x \in \Sigma^*\}$. On sait que $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, mais il est intéressant de trouver des algorithmes pour calculer $\mathcal{L}_I$.

3.4.2 Algorithme énumératif d'interpolation

La définition de $\mathcal{L}_I$, nous permet de construire un algorithme énumératif pour le calculer :
– On construit $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$: pour cela, on énumère tous les préfixes de $\mathcal{L}_D$, soit $|\Sigma| \times \frac{1-|\Sigma|^n}{1-|\Sigma|}$ mots, que l'on peut maximiser par $(n-1) \times |\Sigma|^n$ mots. Et on élimine au fur et à mesure les préfixes de $\mathcal{L}_B$.
– On construit maintenant $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$. Par définiton, les éléments de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ sont des éléments de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$ qui n'ont pas de préfixe dans $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$. On énumère donc tous les mots de $\Omega_{\mathcal{L}_D,\mathcal{L}_B}$, $(n-1) \times |\Sigma|^n$ mots. S'ils ont un préfixe dans $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, on passe au suivant. Sinon, on l'ajoute dans $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, en retirant tous les mots dont il est préfixe. Il nous faut donc parcourir l'ensemble des éléments de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, qui sont au maximum $(n-1) \times |\Sigma|^n$. On a donc $(n-1)^2 \times |\Sigma|^{2n}$ opérations.
– On a maintenant l'ensemble $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, qu'il nous faut compléter pour avoir un langage dans Σ^n. On construit donc pour tous les mots de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, l'ensemble des mots, dont la taille est nécessaire pour compléter ce préfixe. On construit donc dans le pire des cas, $|\Sigma|^n$ mots, pour les $(n-1) \times |\Sigma|^n$ éléments de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$. Soit $(n-1) \times |\Sigma|^{2n}$ opérations.

On a donc un algorithme qui construit $\mathcal{L}_I$. La complexité de cet algorithme est en $\mathcal{O}((n-1)^2 \times |\Sigma|^{2n})$. Ce qui, fait un algorithme exponentiel sur la longueur des mots, et au moins quadratique sur la taille des langages à interpoler (la taille des langages est au maximum de $|\Sigma|^n$). On aurait aussi pu calculer $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$ en utilisant la preuve constructive qu'il est régulier, du théorème 2. Mais l'algorithme décrit ici est un peu plus efficace.

En utilisant une structure de données judicieuse pour représenter les langages, on pourrait réduire la complexité du calcul de l'interpolant. Pour cela, il nous faudrait pouvoir comparer les mots en comparant juste un symbole de plus que le préfixe commun. On pense alors aux structures d'arbre, et aux automates à structure de DAG (Directed Acyclic Graph ou graphe orienté acyclique).

Chapitre 4

Interpolation sur les automates finis à langages dans Σ^n

Dans le cadre du calcul d'un interpolant sur les langages, on a obtenu un algorithme énumératif, et donc peu efficace. Cependant, on souhaite par l'utilisation de structures de données particulières, réduire la complexité du calcul d'un interpolant. Pour cela, la notion que l'on cherche à obtenir de manière efficace, est la notion de préfixe commun. Pour obtenir facilement les préfixes communs de deux langages, on les représente sous forme d'automates finis à structure en DAG. En effet, ces structures ont l'avantage de ne pas dupliquer les préfixes commun d'un langage, et donc de faciliter le travail sur des mots ayant un même préfixe.

4.1 Automates finis à structure en DAG et représentation des langages

Pour pouvoir réduire la complexité du calcul de $\widetilde{\Omega}_{\mathcal{L}_D,\mathcal{L}_B}$, on souhaite représenter les langages par des automates finis déterministes à structure en DAG (Directed Acyclic Graph ou Graphe orienté acyclique).

Définition 8. *[8] Un automate fini déterministe $\mathcal{A}$ est un quintuplet $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ où :*
- *Q est un ensemble fini d'états,*
- *Σ est un alphabet,*
- *$\delta : Q \times \Sigma \to Q$ est la fonction de transition,*
- *$q_0 \in Q$ est l'état initial,*
- *F est l'ensemble des états finaux ou acceptants.*

On peut voir la fonction de transition, δ, comme une fonction indiquant le symbole de l'alphabet à *lire* pour passer d'un état à un autre. A partir de cette définition des automates déterministes, on peut définir le langage accepté par un automate, que l'on notera $\mathcal{L}_\mathcal{A}$. Il nous faut cependant avant définir $\hat{\delta}$ qui est une extension de δ qui ne prend plus un symbole seul, mais une suite de symboles et retourne un état, $\hat{\delta} : Q \times \Sigma^* \to Q$. Cette nouvelle transition correspond à la suite des transitions nécessaires à lire un mot. Le langage d'un automate correspond à l'ensemble des mots acceptés par l'automate, c'est-à-dire, les suites de symboles pour lesquelles il existe une suite de transitions, $\hat{\delta}$, partant de l'état initial et allant jusqu'à un état final.

$$\mathcal{L}_\mathcal{A} = \{w \mid \left(\hat{\delta}(q_0, w) \in F\right)\}$$

Dans notre cas, on restreint les automates pour que le langage accepté par l'automate soit dans Σ^n. Le fait qu'ils ne reconnaissent plus que des mots de Σ^n donnent à leur représentation, une structure en DAG. La figure 4.1 permet de se faire une idée de la façon dont on représente un langage avec un automate à structure en DAG.

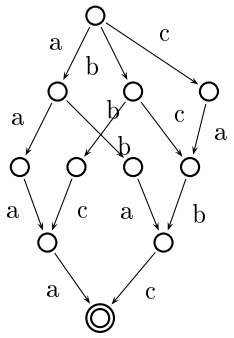


FIGURE 4.1 – Exemple d'automate, $\mathcal{A}_A$ à strucutre en DAG, acceptant le langage $\mathcal{L}_A = \{aaaa, abac, bbca, bcbc, cabc\}$.

Comme les langages acceptés par ces automates, déterministes, à langage dans Σ^n, sont dans les conditions du chapitre précédent (à savoir, des langages de Σ^n), les résultats sont transposables à ces structures de données particulières. De ce fait, à partir de ces structures, on va pouvoir définir un algorithme de calcul d'interpolant en pratique plus efficace que l'algorithme énumératif, se basant sur la définition des langages.

4.2 Comparaison d'automates à structure en DAG

Dans l'optique de calculer notre interpolant, on va donc exploiter la structure de ces automates pour trouver plus facilement les points communs et les préfixes discriminants des deux langages. En fait, ce qui nous intéresse vraiment, c'est la limite entre partie commune et partie différente. Pour trouver la partie commune, il suffit de superposer les automates, ce qui revient à faire un *parcours simultané* des deux automates. Ce parcours simultané sera en fait une extension d'un parcours en profondeur d'un DAG, qui se restreindra à parcourir ce qu'il peut suivre dans les deux automates.

Avant de construire un algorithme, il nous faut définir quelques notations sur les automates à structure en DAG. Pour deux automates $\mathcal{A}_A$ et $\mathcal{A}_B$, on notera $\mathcal{A}_A + \mathcal{A}_B$ l'union des deux automates, qui équivaut à l'union de leurs langages acceptants, $\mathcal{L}_{A+B} = \mathcal{L}_A \cup \mathcal{L}_B$. On note aussi $\mathcal{A}_A.\mathcal{A}_B$ la concaténation de leurs langages acceptants, $\mathcal{L}_{A.B} = \mathcal{L}_A \cap \mathcal{L}_B$. En terme d'automates la concaténation consiste au remplacement de l'état final de A par l'état initial de B. On utilisera, cependant, que la concaténation d'un automate et d'un symbole, c'est-à-dire la concaténation d'un automate et de l'automate acceptant cet unique symbole. On note aussi *Acceptant* un automate ne comportant qu'un état, final, représenté par le quintuplet $(q_0, \Sigma, \varnothing, q_0, q_0)$. Et *Puits* l'automate qui ne permet jamais d'aller dans un état final, même après un certain nombre de transitions, représenté par le quintuplet $(q_0, \Sigma, \varnothing, q_0, \varnothing)$. Pour l'algorithme on a besoin d'une fonction retournant l'automate accessible après la lecture, par l'automate $\mathcal{A}_A$, d'un symbole donné, x, on note cette fonction `branche(A, x)`, si la lecture n'est pas possible, la fonction retourne *Puits*. Par analogie avec les langages, on note Σ l'alphabet entrant de l'automate, et Σ_i le $i^{\text{ème}}$ symbole de l'alphabet. Pour suivre la notation des chapitres précédent, on va comparer un automate $\mathcal{A}_D$ et un automate $\mathcal{A}_B$. Les alphabets de $\mathcal{A}_D$ et de $\mathcal{A}_B$ sont semblables, ainsi que l'alphabet du résultat, noté $\mathcal{A}_C$. Les opérations d'unions et de concaténations ont quelques cas particuliers intéressants :

- L'union d'un automate quelconque, $\mathcal{A}_A$, avec un automate constitué d'un unique état puits, $\mathcal{A}_B = Puits$, donne l'automate $\mathcal{A}_A + Puits = \mathcal{A}_A$.
- La concaténation d'un automate quelconque, $\mathcal{A}_A$, avec un automate constitué d'un unique état puits, $\mathcal{A}_B = Puits$, donne le second automate, $\mathcal{A}_A.Puits = Puits$.
- La concaténation d'un automate quelconque, $\mathcal{A}_A$, avec un automate ne comportant qu'un

état acceptant, $\mathcal{A}_B$ = *Acceptant*, donne le premier automate, $\mathcal{A}_A$.*Acceptant* = $\mathcal{A}_A$.

Algorithme 1 SimultaneousDFS

Entrées: $\mathcal{A}_D$ et $\mathcal{A}_B$ deux automates finis à structure en DAG
Sorties: $\mathcal{A}_C$ un automate montrant les états parcourus

1: **si** ($\mathcal{A}_D \neq$ *Acceptant*) et ($\mathcal{A}_D \neq$ *Puits*) et ($\mathcal{A}_B \neq$ *Acceptant*) et ($\mathcal{A}_B \neq$ *Puits*) **alors**
2: $\mathcal{A}_C$ = *Puits*
3: **pour** $\forall s \in \Sigma$ **faire**
4: $\mathcal{A}_C$ += s . SimultaneousDFS(branche($\mathcal{A}_D$, s), branche($\mathcal{A}_B$, s))
5: **fin pour**
6: **sinon**
7: $\mathcal{A}_C$ = *Puits*
8: **finsi**
9: **retourner** $\mathcal{A}_C$

L'automate $\mathcal{A}_C$ retourné par l'algorithme 1 ne répond pas aux restrictions faites précédemment. En effet, la longueur des mots acceptés est différente en fonction de la longueur des préfixes communs des mots de $\mathcal{A}_D$ et $\mathcal{A}_B$. On verra dans l'algorithme suivant comment faire pour avoir un automate $\mathcal{A}_C$ qui répond à ces restrictions.
L'algorithme 1 étant un parcours en profondeur d'un DAG fini, on sait qu'il termine. On peut voir le résultat de cet algorithme comme le produit synchronisé des automates A et B. Comme les automates que l'on utilise ont une structure en DAG, ils n'ont pas de cycle. Ils sont de plus déterministes. Donc la taille de l'automate résultant n'est pas comme pour un produit cartésien classique $|Q_A| \times |Q_B|$, mais la taille est maximisée par la somme des maxima du nombre de nœuds à chaque profondeur (qui représentera, pour les ROBBD, du chapitre 5, le maximum du nombre de nœuds portant une variable donnée). A partir de cette constatation, on arrive à une taille maximisée par $2 \times max(|Q_A|, |Q_B|)$. Pour se donner une idée de la complexité de cet algorithme, on voit que le parcours est guidé par les automates $\mathcal{A}_D$ et B et que le parcours ne peut pas aller plus loin que le parcours de $\mathcal{A}_D$ (ou de B). On a donc une construction linéaire sur la taille de $\mathcal{A}_D$. De plus, la taille de $\mathcal{A}_D$ est linéaire sur la longueur des mots, à un scalaire près représentant le nombre de mots. Rappelons que l'algorithme énumératif était de complexité au moins quadratique sur la taille des mots.

4.3 Algorithme de calcul d'interpolant sur les automates finis à langage dans Σ^n

On a maintenant un algorithme nous permettant de calculer la partie commune de deux automates finis à structure en DAG. On sait que les automates finis à langages dans Σ^n sont des automates finis à structure en DAG. On peut donc à partir de cet algorithme, rajouter la notion de préfixe discriminant et le calcul d'un interpolant sur les automates à langages dans Σ^n. Cependant, le fait de chercher à calculer un interpolant nécessite que l'on travaille sur des langages disjoints. L'algorithme 2 calcule l'interpolant, $\mathcal{A}_I$, des automates $\mathcal{A}_A$ et $\mathcal{A}_B$. Il a besoin d'un entier représentant la longueur du mot encore à parcourir, servant à compléter les préfixes discriminants avec les mots de la bonne longueur. Cet algorithme utilise alors une fonction `tousMots(k)` qui construit l'automate reconnaissant tous les mots de longueur k sur l'alphabet Σ.

Cet algorithme 2 basé sur l'algorithme 1, présente le même parcours, et donc le même résultat de terminaison. Il nous faut cependant contrôler que cet algorithme calcul bien un interpolant. On va donc vérifier que l'automate C retourné est bien un interpolant de (A, B), pour cela, on va vérifier que $\mathcal{L}_D \subseteq \mathcal{L}_I$ et que $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$. Comme on a vu qu'on pouvait passer simplement d'un automate à un langage, on travaille sur les langages pour simplifier les calculs. Pour prouver cet algorithme, on va faire une preuve par récurrence sur la longueur des mots.

Démonstration.

Algorithme 2 DAGInterpolant

Entrées: $\mathcal{A}_D$ et $\mathcal{A}_B$ deux automates à langage dans Σ^n, tel que $\mathcal{A}_D \cap \mathcal{A}_B = \varnothing$; k un entier représentant la longueur de mot restant à explorer.

Sorties: $\mathcal{A}_I$, un automate à langage dans Σ^n, interpolant $(\mathcal{A}_D, \mathcal{A}_B)$

1: **si** $(\mathcal{A}_D \neq Acceptant)$ et $(\mathcal{A}_D \neq Puits)$ et $(\mathcal{A}_B \neq Acceptant)$ et $(\mathcal{A}_B \neq Puits)$ **alors**
2: $\quad \mathcal{A}_I = Puits$
3: $\quad$ **pour** $\forall s \in \Sigma$ **faire**
4: $\quad\quad \mathcal{A}_I \mathrel{+}= s$. DAGInterpolant(branche($\mathcal{A}_D$, s), branche($\mathcal{A}_B$, s), $k-1$)
5: $\quad$ **fin pour**
6: **sinon**
7: $\quad$ **si** $\mathcal{A}_B = Puits$ et $\mathcal{A}_D \neq Puits$ **alors**
8: $\quad\quad$ // préfixe discriminant
9: $\quad\quad \mathcal{A}_I = tousMots(k)$
10: $\quad$ **sinon**
11: $\quad\quad \mathcal{A}_I = Puits$
12: $\quad$ **finsi**
13: **finsi**
14: **retourner** $\mathcal{A}_I$

Soient $\mathcal{A}_D$ et $\mathcal{A}_B$ deux automates à langage dans Σ^n, avec le même alphabet d'entrée, tel que $\mathcal{L}_D \cap \mathcal{L}_B = \varnothing$. On prouve, pour l'algorithme 2, par induction sur la longueur des mots de $\mathcal{A}_D$ et de $\mathcal{A}_B$ la propriété :

$$\mathcal{L}_D \subseteq \mathcal{L}_I \text{ et } \mathcal{L}_I \cap \mathcal{L}_B = \varnothing \tag{4.1}$$

1er cas, $\mathcal{L}_D = \varnothing$ et $\mathcal{L}_B = \{\epsilon\}$
Le test de la ligne 1 étant faux, et le test de la ligne 7 étant aussi faux, l'algorithme retourne $\mathcal{A}_I = Puits$, et donc $\mathcal{L}_I = \varnothing$, on alors $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$

2ème cas, $\mathcal{L}_D = \{\epsilon\}$ et $\mathcal{L}_B = \varnothing$
Le test de la ligne 1 étant faux, et le test de la ligne 7 étant vrai, l'algorithme retourne $\mathcal{A}_I = tousMots(0)$, et donc $\mathcal{L}_I = \{\epsilon\}$, on alors $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$

3ème cas, $\mathcal{L}_D = \varnothing$ et $\mathcal{L}_B = \varnothing$
Le test de la ligne 1 étant faux, et le test de la ligne 7 étant aussi faux, l'algorithme retourne $\mathcal{A}_I = Puits$, et donc $\mathcal{L}_I = \varnothing$, on alors $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$

La propriété 4.1 est donc vraie pour les automates reconnaissant des mots de longueur 0. On suppose la propriété 4.1 pour les automates dont les mots reconnus sont de longueur n et on prouve que la propriété 4.1 est vraie pour les automates dont les mots reconnus sont de longueur $n+1$.

1er cas, les automates D et B sont différents des automates particuliers.
Le test de la ligne 1 étant vrai, l'algorithme retourne

$$\mathcal{A}_I = \bigdoublebarwedge_{i=1}^{m+1} \Sigma_i.\text{DAGInterpolant}(\text{branche}(\mathcal{A}_D, \Sigma_i), \text{branche}(\mathcal{A}_B, \Sigma_i), n).$$

On pose $\mathcal{A}_{I'_i} = \text{DAGInterpolant}(\text{branche}(\mathcal{A}_D, \Sigma_i), \text{branche}(\mathcal{A}_B, \Sigma_i), n)$. $\forall i \in [1..m+1]$, $\mathcal{A}_{I'_i}$ accepte des mots de longueur n, donc par hypothèse de récurrence on a $\forall i \in [1..m+1]$, $\mathcal{L}_{\text{branche}(\mathcal{A}_D,\Sigma_i)} \subseteq \mathcal{L}_{I'_i}$ et $\mathcal{L}_{I'_i} \cap \mathcal{L}_{\text{branche}(\mathcal{A}_B,\Sigma_i)} = \varnothing$. On a donc $\forall i \in [1..m+1]$, $\Sigma_i.\mathcal{L}_{\text{branche}(\mathcal{A}_D,\Sigma_i)} \subseteq \Sigma_i.\mathcal{L}_{I'_i}$ et $\Sigma_i.\mathcal{L}_{I'_i} \cap \Sigma_i.\mathcal{L}_{\text{branche}(\mathcal{A}_B,\Sigma_i)} = \varnothing$. En ramenant l'union dans les équations, on a $\bigdoublebarwedge_{i=1}^{m+1} \Sigma_i.\mathcal{L}_{\text{branche}(\mathcal{A}_D,\Sigma_i)} \subseteq \bigdoublebarwedge_{i=1}^{m+1} \Sigma_i.\mathcal{L}_{I'_i}$ et $\bigdoublebarwedge_{i=1}^{m+1} \Sigma_i.\mathcal{L}_{I'_i} \cap \bigdoublebarwedge_{i=1}^{m+1} \Sigma_i.\mathcal{L}_{\text{branche}(\mathcal{A}_B,\Sigma_i)} = \varnothing$, les langages unions sont bien disjoints, car les Σ_i sont toujours différents. On peut réécrire ces équations en $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$.

2ème cas, l'automate $\mathcal{A}_D$ est quelconque, et $\mathcal{A}_B = Puits$.
Le test de la ligne 1 est faux, et le test de la ligne 7 est vrai, on a donc $\mathcal{A}_I = tousMots(n+1)$. Il est alors clair que $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$.

3ème cas, l'automate $\mathcal{A}_B$ est quelconque, et $\mathcal{A}_D = Puits$.
Le test de la ligne 1 est faux, et le test de la ligne 7 est faux, l'algorithme nous donne donc $\mathcal{A}_I = Puits$. On a donc sans difficulté $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$.

On a prouvé que si pour les automates reconnaissant des mots de longueur n, $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$, alors pour les automates reconnaissant des mots de longueur $n+1$, $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$. On a aussi prouvé que pour les automates reconnaissant des mots de longueur 0, $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$. On a donc prouvé, par récurrence, que quelque soit la longueur des mots, fixée, $\mathcal{L}_D \subseteq \mathcal{L}_I$ et $\mathcal{L}_I \cap \mathcal{L}_B = \varnothing$.

□

On a maintenant la certitude d'avoir un algorithme, sur les automates à structure en DAG, qui calcule un interpolant en utilisant la notion de préfixe discriminant. A partir de cet algorithme, on va voir dans le chapitre suivant les modifications à faire pour avoir un algorithme de calcul d'interpolant dans le contexte de vérification qui nous intéresse. En effet dans le cadre de la vérification, on cherche à calculer un interpolant entre deux ROBDD.

Chapitre 5

Interpolation sur les ROBDD

Dans le cadre du model-checking symbolique, on utilise souvent les BDD [3, 9] pour représenter les formules de logique propositionnelle. Ces BDD servent alors à représenter la fonction de transition du modèle, et les ensembles d'états. On s'intéresse ici à la représentation des ensembles d'états, et à la recherche d'interpolants de ces ensembles d'états, à l'aide d'algorithmes efficaces exploitant la structure des BDD.

5.1 Introduction aux BDD

On peut se représenter les BDD (Binary Decision Diagram ou Diagramme de décision binaire) comme des automates finis à langage dans Σ^n, et donc à structure en DAG, avec une racine. Ils permettent de représenter une formule de logique propositionnelle, en étiquetant les nœuds du DAG avec les variables libres de la formule, et les arcs sortant d'un nœud par les valuations de sa variable. Les valuations des variables, qui rendent la formule vraie, arrivent dans une feuille Vrai, les autres dans une feuille Faux. La plupart du temps, on travaille avec des BDD ordonnés (OBDD), pour cela, on fixe pour tout le BDD l'ordre dans lequel apparaissent les variables. De plus, on réduit généralement le OBDD, ce qui lui donne une forme canonique, en utilisant deux règles :

- les sous-graphes isomorphes ont une représentation unique,
- les nœuds ayant leurs deux valuations isomorphes sont éliminés.

On parle alors de ROBDD. Dans la suite, on ne parlera que de ROBDD. La figure 5.1 permet de mieux comprendre la structure des ROBDD. Pour simplifier la représentation des BDD, on indique en général que la feuille Vrai. Les arcs allant à la feuille Faux étant sous entendus.

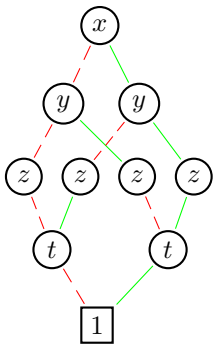


FIGURE 5.1 – Exemple de ROBDD représentant la formule $P = (xz \vee \bar{x}\bar{z}) \wedge (yt \vee \bar{y}\bar{t})$, avec l'ordre x>y>z>t.

On notera la branche représentant la valuation fausse d'une variable, x, par $low(x)$, et par un trait pointillé ou rouge dans les schémas. La valuation vraie sera notée $high(x)$, et représentée par un trait plein ou vert dans les schémas. L'état acceptant de l'automate (*Acceptant* dans le chapitre 4), et l'état puits (*Puits* dans le chapitre 4) sont notées, respectivement, $\top$ et $\bot$, et représentées par un nœud carré avec un 1 ou un 0, aussi nommés feuille Vrai et feuille Faux. On notera $var(x)$ la variable du nœud x, qui notons-le, définit un BDD dont il est la racine. On notera $|A|$ le nombre de nœuds du BDD A, ce nombre de nœuds est égal au nombre de nœuds du DAG, moins le nombre de feuilles Vrai, ou Faux (dans le cadre d'un ROBDD, il n'y a qu'une feuille Vrai et une feuille Faux).

On peut voir les OBDD comme des automates à langage dans $\{0,1\}^n$, et donc comme une application directe de la structure du chapitre 4. Par analogie avec ces strucutres, on peut voir les OBDD comme une restriction des langages.

5.2 Langage accepté par un BDD et préfixes discriminants

Pour simplifier la présentation, on admettra dans la suite que l'ordre des variables des ROBDD est toujours le même.

Le but de cette section est de transposer sur les BDD, et principalement sur les ROBDD, les définitions du chapitre 3. Et donc de voir comment à partir des préfixes discriminants, on peut définir un interpolant, cette fois-si, sur les ROBDD.

Si on parcourt un chemin d'un OBDD, en écrivant la valuation de chaque variable, on obtient un mot sur l'alphabet $\{0,1\}$. Le parcours de tous les chemins du OBDD nous permet de calculer le langage du OBDD. C'est-à-dire, en utilisant l'ordre du OBDD, l'ensemble des conjonctions, de la forme normale disjonctive de la formule que représente le BDD. Si on reprend l'exemple de la figure 5.1, on obtient $\{0000, 0101, 1010, 1111\}$. Ce résultat correspond avec l'ordre du BDD à $\{xyzt, x\bar{y}z\bar{t}, \bar{x}yz\bar{t}, \bar{x}\bar{y}\bar{z}\bar{t}\}$, ce qui nous donne la formule $xyzt \vee x\bar{y}z\bar{t} \vee \bar{x}yz\bar{t} \vee \bar{x}\bar{y}\bar{z}\bar{t}$, qui est bien égale à P. Plus formellement le langage d'un BDD est le langage accepté par l'automate à structure en DAG qui le représente. Il faut, cependant, pour les ROBDD, bien suivre l'ordre des variables, pour pouvoir simuler les variables non présentes, qui ont été supprimées par simplification par la règle 2. L'application de la règle de réduction 2 crée un automate dont la fonction de transition ne lit plus un unique symbole, mais une séquence de symboles.
On obtient donc, à partir d'un BDD, un langage, de mots de longueur finie, sur l'alphabet $\{0,1\}$. On notera le langage du BDD A, $\mathcal{L}_A$. A partir de cette définition du langage des BDD, on a envie de transposer la relation d'ordre des langages sur les BDD, en apportant quelques ajouts induits par la représentation par les BDD de formules de logique.

Définition 9.
Soient A et B deux ROBDD. Par notation,

$$A \sqsubseteq B \iff \mathcal{L}_A \subseteq \mathcal{L}_B \iff A \vee B = B \iff A \wedge B = A$$

Cette transformation des BDD en langages, nous permet aussi de réutiliser toutes les définitions du chapitre 3. Et en particulier le calcul d'un interpolant entre deux langages disjoints. Réécrivons quand même la définition des préfixes discriminants, mais pour cela, il faut définir ce qu'est un préfixe dans un BDD. On parlera de préfixe d'un nœud, qui correspond à un chemin possible permettant d'arriver dans ce nœud, c'est donc la suite des valuations nécessaires pour arriver dans ce nœud. Si w est préfixe du langage de B alors $B \wedge w \neq \bot$.

Définition 10 (préfixe discriminant sur les ROBDD)**.**
Soit B un ROBDD, alors le préfixe w est un préfixe discriminant B si et seulement si $B \wedge w = \bot$.

La définition d'un préfixe discriminant se transpose très bien des langages aux BDD. On peut alors bien calculer l'ensemble des préfixes discriminants, $\Omega_{D,B} = \{w \mid (D \wedge w \neq \bot \text{ et } B \wedge w = \bot)\}$, et l'ensemble des plus petits préfixes discriminants $\widetilde{\Omega}_{D,B}$. Pour ce dernier, il nous faut cependant transposer la relation d'ordre préfixe sur les BDD, simplement en considérant un préfixe comme un mot sur $\{0,1\}$. Avec cette définition, et avec un raisonnement similaire à celui du chapitre 3, on obtient le même interpolant en partant de $\widetilde{\Omega}_{D,B}$. Et donc le résultat du théorème 3 est aussi transposable.

Proposition 2.
Soient D et B deux ROBDD,

$$D \Longrightarrow \widetilde{\Omega}_{D,B} \text{ et } \widetilde{\Omega}_{D,B} \wedge B = \bot.$$

On cherche maintenant à savoir si l'interpolant que l'on vient de calculer présente un intérêt pour notre problématique. On a donc défini un critère nous permettant d'évaluer la qualité de notre interpolant. Pour cela, on a choisi de dire qu'un ROBDD est meilleur qu'un autre, si son langage est plus grand, et son nombre de nœuds plus petit. Pour cela, on a défini la relation d'ordre suivante, différente de la relation d'ordre de la définition 9.

Définition 11.
Soit A et B deux ROBDD tels que

$$A \preccurlyeq B \Longleftrightarrow B \sqsubseteq A \text{ et } |A| \leq |B|$$

On veut appliquer cette relation d'ordre sur un BDD interpolant, I, et sur le BDD interpolé, A. On a donc par définition d'un interpolant $A \sqsubseteq I$. On s'intéresse donc qu'aux BDD comparables 2 à 2 par $\sqsubseteq$.

Proposition 3.
Soient A et B deux ROBDD, comparables par $\sqsubseteq$.

La relation $\preccurlyeq$, sur A et B, est une relation d'ordre total.

La preuve de cette relation d'ordre est triviale. En effet, on se restreint aux ROBDD comparables par $\sqsubseteq$, donc la relation d'ordre se réduit alors à $\leq$. Et la relation $\leq$ est une relation d'ordre totale.

5.3 Algorithmes de calcul d'interpolant sur les ROBDD

On a vu que les OBDD ont des structures très proches des automates finis à structure en DAG du chapitre 4. On va donc pouvoir réutiliser l'algorithme de calcul d'interpolant que l'on a prouvé à la fin de ce chapitre précédent. On va cependant être obligé de faire quelques modifications pour pouvoir gérer les cas induits par la règle de réduction des OBDD en ROBDD suivante : "les nœuds ayant leurs deux valuations isomorphes sont éliminés". Cependant, l'alphabet très particulier des BDD va nous permettre de simplifier l'algorithme à certains endroits. Si on reprend l'algorithme 2 et qu'on le transpose aux BDD, on obtient l'algorithme 3. On observe que la boucle *pour* qui permettait de parcourir tout l'alphabet a pu être remplacée par deux cas, un pour chaque symbole de l'alphabet de BDD, $\{0,1\}$. On a aussi remplacé la concaténation, par l'operateur $\wedge$ et l'union par l'opérateur $\vee$, qui ont le même sens pour les BDD. En effet, la concaténation équivaut à fixer le comportement de certaines variables. Et si on prend l'union de deux langages, $\mathcal{L}_D \cup \mathcal{L}_B$, cela correpond aux mots de $\mathcal{L}_D$ ou au mots de $\mathcal{L}_B$, et donc bien à un ou logique sur la structure les représentants.

Comme on l'a dit précédemment, cet algorithme ne travaille que sur les OBDD, il nous faut mettre en place une règle de *simulation de variable* pour qu'il puisse fonctionner sur les ROBDD. Cette règle consistera à faire l'inverse de la deuxième règle de réduction, dans le but de l'annuler.

Algorithme 3 OBDDInterpolant

Entrées: D et B deux OBDD, tels que $D \wedge B = \bot$, k un entier représentant la longueur de mot restant à explorer.

Sorties: I un OBDD interpolant (D, B)

```
1:  si (D ≠ ⊥) et (D ≠ ⊤) et (B ≠ ⊥) et (B ≠ ⊤) alors
2:     I = (¬var(B) ∧ OBDDInterpolant(low(D), low(B), k − 1))
3:           ∨ (var(B) ∧ OBDDInterpolant(high(D), high(B), k − 1))
4:  sinon
5:     si (B = ⊥ et D ≠ ⊥) alors
6:        // préfixe discriminant
7:        I = tousMots(k)
8:     sinon
9:        I = ⊥
10:    finsi
11: finsi
12: retourner I
```

Et donc si on se trouve avec deux ROBDD ayant pour racine deux variables différentes, alors la variable la plus grande dans l'ordre des variables des ROBDD, peut être considérée comme ayant une valeur indifférente (vraie ou fausse). On voit aussi que le fait de rajouter tous les mots de longueur k (par la fonction `tousMots(k)`) peut être simplifié dans le cas des ROBDD. En effet, cela consiste à dire que toutes les variables, suivant la variable du nœud courant, sont vraies ou fausses. Donc par l'application de la deuxième règle de réduction, on peut enlever ces variables, et donc avoir une transition qui va directement dans la feuille Vrai. On obtient alors l'algorithme 4.

Pour mieux comprendre les résultats fournis par cet algorithme, et par les algorithmes, optimisés, suivants, on va analyser un exemple. Il est issu d'un article de McMillan [10]. Cet exemple correspond exactement au contexte dans lequel on cherche à construire ces interpolants. C'est-à-dire trouver un interpolant entre les états puits, et les états néfastes de la concrétisation d'un état de faute d'un contre-exemple fallacieux (introduit dans la section 2.2). On a donc les ensembles S_D et S_B issus du deuxième raffinement de l'analyse d'un modèle. On a juste fait une modification, pour pouvoir utiliser cet exemple sur tous les algorithmes. On a créé un cas nous permettant d'exploiter les règles de simulation de variable, en retirant de S_D la nécessité que la variable b soit vraie.

D'après les ROBDD de la figure 5.2, on a $S_D = \bar{a}b\big(\bar{c}\bar{d}\bar{e}(fgh \vee \bar{f}\bar{g}h) \vee cd\bar{e}\bar{f}\bar{g}h\big)$ et $S_B = \bar{a}\Big(\bar{c}\big(\bar{d}\bar{e}(\bar{f}gh \vee f\bar{g}h) \vee d\bar{e}\bar{g}h\big) \vee c\bar{d}\bar{e}\bar{f}h\Big)$, et on cherche S'_D, un interpolant de (S_D, S_B).

La terminaison de cet algorithme 4 est toujours assurée, par le fait qu'on effectue un parcours en profondeur d'un DAG fini. Mais on peut se demander si le résultat retourné est bien un interpolant de ses deux paramètres. Pour cela, et de manière similaire à la preuve de l'algorithme 2, on va faire une preuve par récurrence, sur le nombre de variables des BDD D et B.

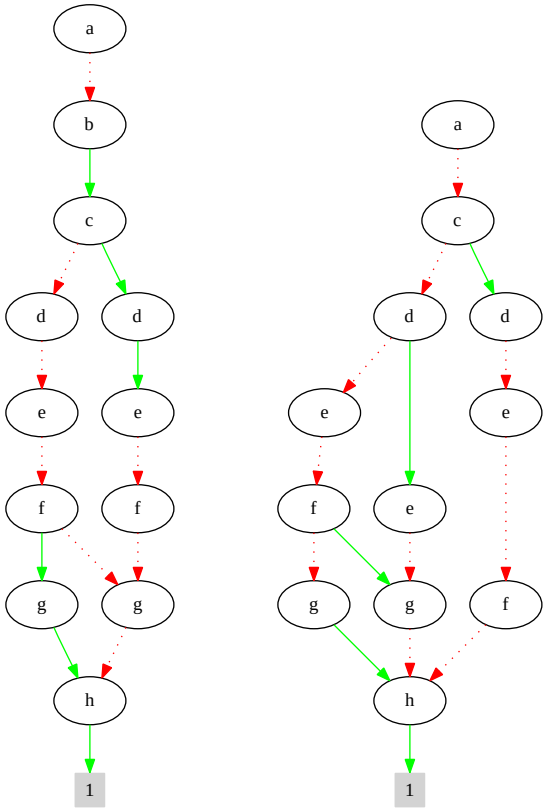
Démonstration.
Soient D et B deux ROBDD, sur un ensemble $X = \{x_1, ..., x_n\}$ de variables, tels que l'ordre des variables pour les ROBDD soit $x_n > .. > x_1$. On prouve par induction sur $|X|$ la propriété, caractérisant un interpolant,

$$D \Longrightarrow I \text{ et } B \wedge I = \bot \tag{5.1}$$

Commençons par le cas où $|X| = 0$: (trois cas possibles, tels que $D \wedge B = \bot$)

1er cas, $D = \bot$ et $B = \top$
Les tests des lignes 1 et 16 sont faux, l'algorithme retourne donc $I = \bot$, et il est évident que $I \wedge B = \bot$ et $D \Rightarrow I$.

2ème cas, $D = \top$ et $B = \bot$
Le test de la ligne 1 est faux, mais le test de la ligne 16 est vrai, l'algorithme retourne donc $I = \top$, et il est évident que $I \wedge B = \bot$ et $D \Rightarrow I$.



(a) S_D (b) S_B

FIGURE 5.2 – Exemple de McMillan

3ème cas, $D = \bot$ et $B = \bot$
Les tests des ligne 1 et 16 sont faux, l'algorithme retourne donc $I = \bot$, et il est évident que $I \wedge B = \bot$ et $D \Rightarrow I$.

Le quatrième cas, $D = \top$ et $B = \top$, alors $D \wedge B \neq \varnothing$, les hypothèses ne sont donc pas satisfaites. On a donc bien, pour $|X| = 0$, si $D \wedge B = \bot$ alors $I \wedge B = \bot$ et $D \Rightarrow I$.
Supposons maintenant que pour $|X| = n$, l'hypothèse $I \wedge B = \bot$ et $D \Rightarrow I$ est vérifiée. On prouve que la propriété 5.1 est vraie pour $|X| = n + 1$ (6 cas possibles, tels que $D \wedge B = \bot$).

1er cas, la variable x_{n+1} est racine de D et de B
Les tests des lignes 1 et 2 sont vrais, on a donc

$$I = (\text{ROBDDInterpolant}(low(D), low(B)) \wedge \overline{x_{n+1}}) \vee (\text{ROBDDInterpolant}(high(D), high(B)) \wedge x_{n+1})$$

On pose
– $I_1 = \text{ROBDDInterpolant}(low(D), low(B))$,
– $I_2 = \text{ROBDDInterpolant}(high(D), high(B))$,
on obtient donc $I = (I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})$
Comme D et B sont disjoints, $(D \wedge x_{n+1}) \wedge (B \wedge x_{n+1}) = \bot$ et donc $high(D) \wedge high(B) = \bot$. De manière similaire on obtient $low(D) \wedge low(B) = \bot$. On peut donc appliquer l'algorithme de calcul d'interpolant. $low(D)$, $low(B)$, $high(D)$, $high(B)$ ne comportent plus que n variables, sans x_{n+1}, on a donc par hypothèse de récurrence :
– $I_1 \wedge low(B) = \bot$ et $low(D) \Rightarrow I_1$
– $I_2 \wedge high(B) = \bot$ et $high(D) \Rightarrow I_2$
Montrons que $I \wedge B = \bot$:
$I_1 \wedge low(B) = \bot$ implique que $I_1 \wedge \overline{x_{n+1}} \wedge low(B) \wedge \overline{x_{n+1}} = \bot$ et $I_2 \wedge high(B) = \bot$ implique que $I_2 \wedge$

Algorithme 4 ROBDDInterpolant

Entrées: D et B deux ROBDD, tels que $D \wedge B = \bot$
Sorties: I un ROBDD interpolant (D, B)

```
1: si (D ≠ ⊥) et (D ≠ ⊤) et (B ≠ ⊥) et (B ≠ ⊤) alors
2:    si var(D) = var(B) alors
3:       I = (ROBDDInterpolant(low(D), low(B)) ∧ ¬var(D))
4:           ∨(ROBDDInterpolant(high(D), high(B)) ∧ var(D))
5:    sinon
6:       // Simulation de variable
7:       si var(D) < var(B) alors
8:          I = (ROBDDInterpolant(D, low(B)) ∧ ¬var(B))
9:              ∨(ROBDDInterpolant(D, high(B)) ∧ var(B))
10:      sinon
11:         I = (ROBDDInterpolant(low(D), B) ∧ ¬var(D))
12:             ∨(ROBDDInterpolant(high(D), B) ∧ var(D))
13:      finsi
14:   finsi
15: sinon
16:   si (B = ⊥ et D ≠ ⊥) alors
17:      // préfixe discriminant
18:      I = ⊤
19:   sinon
20:      I = ⊥
21:   finsi
22: finsi
23: retourner I
```

$x_{n+1} \wedge high(B) \wedge x_{n+1} = \bot$. On a donc $(I_1 \wedge \overline{x_{n+1}} \wedge low(B) \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1} \wedge high(B) \wedge x_{n+1}) = \bot$, que l'on peut réécrire en $((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})) \wedge ((low(B) \wedge \overline{x_{n+1}}) \vee (high(B) \wedge x_{n+1})) = \bot$. Avec ce résultat, on a $I \wedge B = \bot$.
Montrons maintenant que $D \Rightarrow I$:
$low(D) \Rightarrow I_1$ implique $(low(D) \wedge \overline{x_{n+1}}) \Rightarrow (I_1 \wedge \overline{x_{n+1}})$ et $high(D) \Rightarrow I_2$ implique $(high(D) \wedge x_{n+1}) \Rightarrow (I_2 \wedge x_{n+1})$. A partir de ces deux résultats, on a $((low(D) \wedge \overline{x_{n+1}}) \vee (high(D) \wedge x_{n+1})) \Rightarrow ((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1}))$. Et donc $D \Rightarrow I$.

2ème cas, la variable x_{n+1} est racine de D mais pas de B
Le test de la ligne 1 est vrai, mais le test de la ligne 2 est faux, on se trouve alors dans le cas d'une simulation de variable. On a aussi le test de la ligne 7 qui est faux, et donc l'algorithme retourne

$$I = (\text{ROBDDInterpolant}(low(D), B) \wedge \overline{x_{n+1}}) \vee (\text{ROBDDInterpolant}(high(D), B) \wedge x_{n+1})$$

On pose
– $I_1 = \text{ROBDDInterpolant}(low(D), B)$,
– $I_2 = \text{ROBDDInterpolant}(high(D), B)$,
on obtient donc $I = (I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})$
$high(D) \wedge B = \bot$ et $low(D) \wedge B = \bot$, on peut donc appliquer l'algorithme de calcul d'interpolant. $low(D)$, $high(D)$, B ne comportent plus que n variables, sans x_{n+1}, on a donc par hypothèse de récurrence :
– $I_1 \wedge B = \bot$ et $low(D) \Rightarrow I_1$
– $I_2 \wedge B = \bot$ et $high(D) \Rightarrow I_2$
Montrons que $I \wedge B = \bot$:
$I_1 \wedge B = \bot$ implique que $I_1 \wedge \overline{x_{n+1}} \wedge B \wedge \overline{x_{n+1}} = \bot$ et $I_2 \wedge B = \bot$ implique que $I_2 \wedge x_{n+1} \wedge B \wedge x_{n+1} = \bot$. On a donc $(I_1 \wedge \overline{x_{n+1}} \wedge B \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1} \wedge B \wedge x_{n+1}) = \bot$, que l'on peut réécrire en $((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})) \wedge ((B \wedge \overline{x_{n+1}}) \vee (B \wedge x_{n+1})) = \bot$. Avec ce résultat, on a $I \wedge B = \bot$.

Montrons maintenant que $D \Rightarrow I$:
$low(D) \Rightarrow I_1$ implique $(low(D) \wedge \overline{x_{n+1}}) \Rightarrow (I_1 \wedge \overline{x_{n+1}})$ et $high(D) \Rightarrow I_2$ implique $(high(D) \wedge x_{n+1}) \Rightarrow (I_2 \wedge x_{n+1})$. A partir de ces deux résultats, on a $((low(D) \wedge \overline{x_{n+1}}) \vee (high(D) \wedge x_{n+1})) \Rightarrow ((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1}))$. Et donc $D \Rightarrow I$.

3ème cas, la variable x_{n+1} est racine de B mais pas de D
Comme dans le cas précédent on a le test de la ligne 1 vrai, et le test de la ligne 2 faux, mais le test de la ligne 7 est cette fois vrai. L'algorithme retourne donc

$$I = (\text{ROBDDInterpolant}(D, low(B)) \wedge \overline{x_{n+1}}) \vee (\text{ROBDDInterpolant}(D, high(B)) \wedge x_{n+1})$$

On pose
– $I_1 = \text{ROBDDInterpolant}(D, low(B))$,
– $I_2 = \text{ROBDDInterpolant}(D, high(B))$,
on obtient donc $I = (I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})$
$D \wedge high(B) = \bot$ et $D \wedge low(B) = \bot$, on peut donc appliquer l'algorithme de calcul d'interpolant. D, $low(B)$, $high(B)$ ne comportent plus que n variables, sans x_{n+1}, on a donc par hypothèse de récurrence :
– $I_1 \wedge low(B) = \bot$ et $D \Rightarrow I_1$
– $I_2 \wedge high(B) = \bot$ et $D \Rightarrow I_2$
Montrons que $I \wedge B = \bot$:
$I_1 \wedge low(B) = \bot$ implique que $I_1 \wedge \overline{x_{n+1}} \wedge low(B) \wedge \overline{x_{n+1}} = \bot$ et $I_2 \wedge high(B) = \bot$ implique que $I_2 \wedge x_{n+1} \wedge high(B) \wedge x_{n+1} = \bot$. On a donc $(I_1 \wedge \overline{x_{n+1}} \wedge low(B) \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1} \wedge high(B) \wedge x_{n+1}) = \bot$, que l'on peut réécrire en $((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1})) \wedge ((low(B) \wedge \overline{x_{n+1}}) \vee (high(B) \wedge x_{n+1})) = \bot$. Avec ce résultat, on a $I \wedge B = \bot$.
Montrons maintenant que $D \Rightarrow I$:
$D \Rightarrow I_1$ implique $(D \wedge \overline{x_{n+1}}) \Rightarrow (I_1 \wedge \overline{x_{n+1}})$ et $D \Rightarrow I_2$ implique $(D \wedge x_{n+1}) \Rightarrow (I_2 \wedge x_{n+1})$. A partir de ces deux résultats, on a $((D \wedge \overline{x_{n+1}}) \vee (D \wedge x_{n+1})) \Rightarrow ((I_1 \wedge \overline{x_{n+1}}) \vee (I_2 \wedge x_{n+1}))$. Et donc $D \Rightarrow I$.

4ème cas, la variable x_{n+1} n'est racine ni de D ni de B
On est ici, dans le cas de BDD avec n variables, on a donc par hypothèse de récurrence, $I \wedge B = \bot$ et $D \Rightarrow I$.

5ème cas, la variable x_{n+1} est racine de D, mais $B = \bot$
Le test de la ligne 1 est faux, on va donc à la ligne 16 dont le test est vrai. Dans ce cas, l'algorithme produit directement $I = \top$. Et on a bien $I \wedge B = \bot$, car $B = \bot$, et $D \Rightarrow I$.

6ème cas, la variable x_{n+1} est racine de B, mais $D = \bot$
Dans ce cas, le test des lignes 1 et 16 sont faux, l'algorithme produit donc $I = \bot$. Et on a bien $I \wedge B = \bot$, car $I = \bot$, et $D \Rightarrow I$.

On a donc pour tout les cas à $n+1$ variables $I \wedge B = \bot$ et $D \Rightarrow I$. Donc par récurrence, $I \wedge B = \bot$ et $D \Rightarrow I$ quelque soit le nombre de variables, fixé, des BDD D et B. □

On a maintenant un algorithme qui calcule un interpolant entre deux ROBDD. On va maintenant essayer, en conservant le fait que la sortie soit un interpolant, de réduire la taille du ROBDD résultat.

En revenant sur notre exemple fil rouge, le calcul de l'interpolant de (S_D, S_B) par l'algorithme 4 donne le ROBDD de la figure 5.3. Si on analyse le résultat, le ROBDD S_D comporte 12 nœuds, et son langage contient 3 mots, le ROBDD S_B comporte lui aussi 12 nœuds. L'interpolant obtenu ne comporte quant à lui que 9 nœuds, il est donc bien plus petit (par la définition 11) que S_D ou S_B, de plus son langage contient 20 mots. Cependant, on observe que sur les S_D, S_B, et S'_D les variables a et e sont toujours fausses. Avec cette remarque, on crée la règle de réduction de variable qui suit.

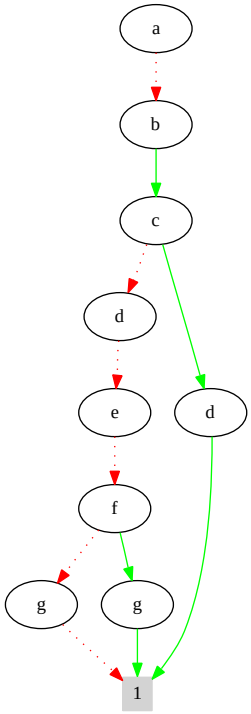


FIGURE 5.3 – Exemple de McMillan 5.2, calcul d'un interpolant par l'algorithme 4.

5.3.1 Règle de réduction de variables

Pour réduire la taille du résultat, en terme de nombre de nœuds, on va s'intéresser au cas particulier, où un préfixe n'appartient ni à un langage, ni à l'autre. Pour situer ce cas, dans notre algorithme, on peut le voir comme le fait, qu'après un préfixe commun, une valuation de la variable courante, amène directement dans la feuille Faux, dans les deux ROBDD. Le fait que tous les mots commençant par ce préfixe ne soient inclus dans aucun des deux langages (langage associé à un ROBDD) nous permet de dire que l'on peut rajouter tous ces mots à l'interpolant que l'on calcule. En termes de ROBDD, cela ne va pas réduire le nombre de nœuds, de faire aller la valuation d'une variable dans la feuille Vrai ou dans la feuille Faux. Cependant, on sait par la deuxième règle de réduction, que si un nœud a ces deux valuations ayant le même sous-graphe alors on peut éliminer ce nœud. On a donc envie d'appliquer cette règle pour éliminer des nœuds. On va donc pouvoir dans notre cas, ne pas rajouter toutes les possibilités mais seulement celles de l'autre valuation de la variable. Et donc amener les deux valuations de ce nœud dans le même sous-arbre, et ainsi provoquer l'élimination de ce nœud. A partir de cette constatation, on peut écrire un nouvel algorithme 5, se basant sur l'algorithme 4.

En appliquant ce nouvel algorithme 5 sur le calcul d'interpolant de (S_D, S_B), de notre exemple fil rouge, on arrive au ROBDD de la figure 5.4. Si on regarde la taille de ce ROBDD, on constate qu'il ne contient que 7 nœuds. Il est donc plus petit que le ROBDD résultant de l'algorithme précedent.

5.3.2 Règle de simulation de variables par "Craig"

On a au moment d'écrire un algorithme permettant de calculer un interpolant sur les ROBDD, parlé du problème de simulation de variable. En effet, dans certain cas, on cherche à comparer des ROBDD n'ayant pas à leurs racines la même variable. On a alors introduit un règle de *simulation de variable* qui nous permet de traiter ce cas, en rajoutant dans l'interpolant (et en la simulant dans le ROBDD où elle est absente) la variable en question.

Algorithme 5 ROBDDInterpolantReduced

Entrées: D et B deux ROBDD, tels que $D \wedge B = \bot$
Sorties: I un ROBDD interpolant (D, B)

si $(D \neq \bot)$ et $(D \neq \top)$ et $(B \neq \bot)$ et $(B \neq \top)$ **alors**
  si $var(D) = var(B)$ **alors**
    si $(low(D) = \bot$ et $low(B) = \bot)$ ou $(high(D) = \bot$ et $high(B) = \bot)$ **alors**
      // Simplification de variable
      I = ROBDDInterpolantReduced$(low(D), low(B))$
        $\vee$ ROBDDInterpolantReduced$(high(D), high(B)$
    sinon
      $I = \big($ROBDDInterpolantReduced$(low(D), low(B)) \wedge \overline{var(D)}\big)$
        $\vee\big($ROBDDInterpolantReduced$(high(D), high(B)) \wedge var(D)\big)$
    finsi
  sinon
    // Simulation de variable
    si $var(D) < var(B)$ **alors**
      $I = \big($ROBDDInterpolantReduced$(D, low(B)) \wedge \overline{var(B)}\big)$
        $\vee\big($ROBDDInterpolantReduced$(D, high(B)) \wedge var(B)\big)$
    sinon
      $I = \big($ROBDDInterpolantReduced$(low(D), B) \wedge \overline{var(D)}\big)$
        $\vee\big($ROBDDInterpolantReduced$(high(D), B) \wedge var(D)\big)$
    finsi
  finsi
sinon
  si $(B = \bot$ et $D \neq \bot)$ **alors**
    // préfixe discriminant
    $I = \top$
  sinon
    $I = \bot$
  finsi
finsi
retourner D

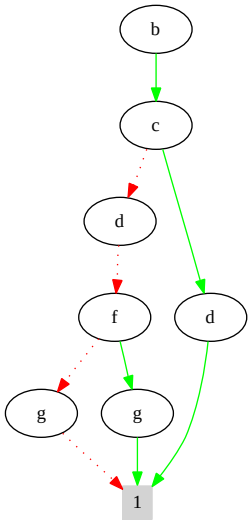


FIGURE 5.4 – Exemple de McMillan 5.2, calcul d'un interpolant par l'algorithme 5.

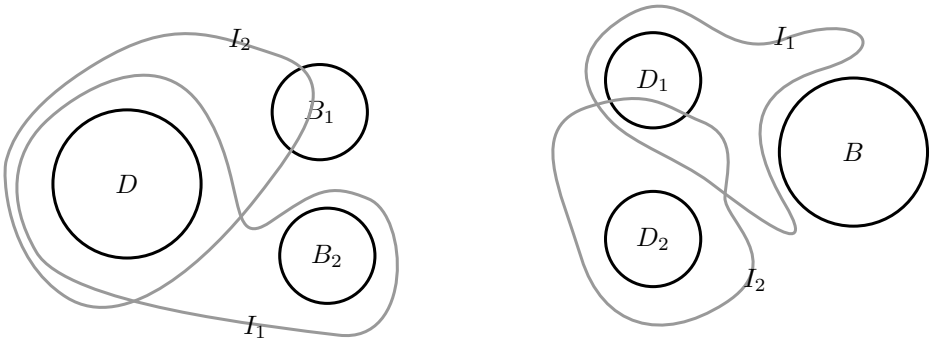
Cependant, Craig, dans [6], a introduit le fait qu'un interpolant entre deux formules de logique, ne

comportait nécessairement que les variables présentes dans les deux formules à interpoler. Cette notion d'interpolant de Craig, peut être transposée aux BDD. Pour cela, on note $\mathcal{V}ar(A)$ les variables présentes dans le ROBDD A.

Définition 12 (Interpolant de Craig)**.** *Soient D et B deux ROBDD, tels que $D \wedge B = \bot$, alors I est un interpolant de Craig, si et seulement si*

$$D \Longrightarrow I \text{ et } B \wedge I = \bot \text{ et } \mathcal{V}ar(I) = \mathcal{V}ar(D) \cap \mathcal{V}ar(B)$$

On peut donc créer une nouvelle règle de *simulation de variable*, qui respecte l'interpolant de Craig. Il nous faut cependant disjoindre les deux cas, soit la variable n'est pas présente dans D, soit la variable n'est pas présente dans B. A partir de ces deux cas, on va pouvoir se donner une intuition de la façon de définir un nouvel interpolant respectant les conditions de l'interpolant de Craig. On va prendre les langages des sous-ROBDD. C'est à dire dans le premier cas le langage de D, $\mathcal{L}_D$, et les langages de $low(B)$, $\mathcal{L}_{B_1}$, et de $high(B)$, $\mathcal{L}_{B_2}$. Si on prend un interpolant, I_1, de $(\mathcal{L}_D, \mathcal{L}_{B_1})$ et un interpolant, I_2, de $(\mathcal{L}_D, \mathcal{L}_{B_2})$, comme sur la figure 5.5(a). On voit que pour avoir un interpolant qui n'intersecte ni $\mathcal{L}_{B_1}$ ni $\mathcal{L}_{B_2}$, on peut prendre l'intersection des deux interpolants, I_1 et I_2. Pour le deuxième, on prend le langage de B, $\mathcal{L}_B$, et les langages de $low(D)$, $\mathcal{L}_{D_1}$, et de $high(D)$, $\mathcal{L}_{D_2}$. Si on prend un interpolant, I_1, de $(\mathcal{L}_{D_1}, \mathcal{L}_B)$ et un interpolant, I_2, de $(\mathcal{L}_{D_2}, \mathcal{L}_B)$, comme sur la figure 5.5(b). On voit que pour avoir un interpolant qui contienne $\mathcal{L}_{D_1}$ et $\mathcal{L}_{D_2}$, on peut prendre l'union des deux interpolants, I_1 et I_2.



(a) Variable absente dans D (b) Variable absente dans B

FIGURE 5.5 – Interpolant Craig

Avec l'intuition donnée par la figure 5.3.2, on écrit l'algorithme 6.

Démonstration. On ne va pas redémontrer tout l'algorithme qui est basé sur l'algorithme 4. On va juste se focaliser sur le fait qu'après une simulation de variable "par Craig" on a bien un interpolant. Pour cela, on va travailler sur les langages des ROBDD. On va cependant conserver la même notation que dans la preuve de l'algorithme 4. Soient D et B deux ROBDD, sur un ensemble $X = \{x_1, ..., x_n\}$ de variables, tel que l'ordre des variables pour les ROBDD soit $x_n > .. > x_1$.

1er cas , la variable X_n est présente dans D, mais pas dans B. Le test de la ligne 1 est faux, et le test de la ligne 7 est vrai. On a donc I = ROBDDCraigInterpolant$(low(D), B) \vee$ ROBDDCraigInterpolant$(high(D), B)$. On pose
- I_1 = ROBDDCraigInterpolant$(low(D), B)$,
- I_2 = ROBDDCraigInterpolant$(high(D), B)$,

En passant aux langages, on a $\mathcal{L}_I = x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1} \cup x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2}$. On note B_0 tel que $\forall v \in \{0,1\}, \mathcal{L}_{v.B_0} = \mathcal{L}_B$, comme la variable x_n n'apparaît pas dans B, le ROBDD de B_0 est le même que celui de B, en considérant qu'on ne travaille plus que sur $n-1$ variables.
Montrons que $\mathcal{L}_D \subseteq \mathcal{L}_I$ et donc que $D \Rightarrow I$:
Comme I_1 est un interpolant de $(low(D), B_0)$, on a $\mathcal{L}_{low(D)} \subseteq \mathcal{L}_{I_1}$ et donc $x_n.\mathcal{L}_{low(D)} \subseteq$

Algorithme 6 ROBDDCraigInterpolant

Entrées: D et B deux ROBDD, tels que $D \wedge B = \bot$
Sorties: I un ROBDD interpolant, de Craig, de(D, B)

```
1: si (D ≠ ⊥) et (D ≠ ⊤) et (B ≠ ⊥) et (B ≠ ⊤) alors
2:    si var(D) = var(B) alors
3:       I = (ROBDDCraigInterpolant(low(D), low(B)) ∧ ¬var(B))
4:           ∨ (ROBDDCraigInterpolant(high(D), high(B)) ∧ var(B))
5:    sinon
6:       // Simulation de variable "par Craig"
7:       si var(D) < var(B) alors
8:          I = ROBDDCraigInterpolant(D, low(B))
9:              ∧ ROBDDCraigInterpolant(D, high(B))
10:      sinon
11:         I = ROBDDCraigInterpolant(low(D), B)
12:             ∨ ROBDDCraigInterpolant(high(D), B)
13:      finsi
14:   finsi
15: sinon
16:   si (B = ⊥ et D ≠ ⊥) alors
17:      // préfixe discriminant
18:      I = ⊤
19:   sinon
20:      I = ⊥
21:   finsi
22: finsi
23: retourner I
```

$x_n.\mathcal{L}_{I_1}$ et $\overline{x_n}.\mathcal{L}_{low(D)} \subseteq \overline{x_n}.\mathcal{L}_{I_1}$. Ce qui transposable pour I_2, en $x_n.\mathcal{L}_{high(D)} \subseteq x_n.\mathcal{L}_{I_2}$ et $\overline{x_n}.\mathcal{L}_{high(D)} \subseteq \overline{x_n}.\mathcal{L}_{I_2}$. En passant à l'union, on obtient $\big((x_n.\mathcal{L}_{low(D)}) \cup (\overline{x_n}.\mathcal{L}_{low(D)}) \cup (x_n.\mathcal{L}_{high(D)}) \cup (\overline{x_n}.\mathcal{L}_{high(D)})\big) \subseteq \big((x_n.\mathcal{L}_{I_1}) \cup (\overline{x_n}.\mathcal{L}_{I_1}) \cup (x_n.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{I_2})\big)$. En reprenant la définition de $\mathcal{L}_I$ et de $\mathcal{L}_D$, on a $\big(\mathcal{L}_D \cup (x_n.\mathcal{L}_{low(D)}) \cup (\overline{x_n}.\mathcal{L}_{high(D)})\big) \subseteq \mathcal{L}_I$, et donc $\mathcal{L}_D \subseteq \mathcal{L}_I$, en revenant aux ROBDD, on a $D \Rightarrow I$.
Montrons que $\mathcal{L}_B \cap \mathcal{L}_I = \varnothing$ et donc que $B \wedge I = \bot$:
Comme I_1 est un interpolant de $(low(D), B_0)$, on a $\mathcal{L}_{B_0} \cap \mathcal{L}_{I_1} = \varnothing$ et donc $x_n.\mathcal{L}_{B_0} \cap x_n.\mathcal{L}_{I_1} = \varnothing$ et $\overline{x_n}.\mathcal{L}_{B_0} \cap \overline{x_n}.\mathcal{L}_{I_1} = \varnothing$. Ce qui transposable pour I_2, en $x_n.\mathcal{L}_{B_0} \cap x_n.\mathcal{L}_{I_2} = \varnothing$ et $\overline{x_n}.\mathcal{L}_{B_0} \cap \overline{x_n}.\mathcal{L}_{I_2} = \varnothing$. En faisant l'union de ces équations, $(x_n.\mathcal{L}_{B_0} \cap x_n.\mathcal{L}_{I_1}) \cup (\overline{x_n}.\mathcal{L}_{B_0} \cap \overline{x_n}.\mathcal{L}_{I_1}) \cup (x_n.\mathcal{L}_{B_0} \cap x_n.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{B_0} \cap \overline{x_n}.\mathcal{L}_{I_2}) = \varnothing$ et après une petite phase calculatoire, on arrive à $(x_n.\mathcal{L}_{B_0} \cup \overline{x_n}.\mathcal{L}_{B_0}) \cap (x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1} \cup x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2}) = \varnothing$. En utilisant les définitions de $\mathcal{L}_B$ et de $\mathcal{L}_I$, on a $\mathcal{L}_B \cap \mathcal{L}_I = \varnothing$ et donc que $B \wedge I = \bot$.

2ème cas , la variable X_n est présente dans B, mais pas dans D. Le test de la ligne 1 est faux, et le test de la ligne 7 est faux. On a donc I = ROBDDCraigInterpolant$(D, low(B)) \wedge$ ROBDDCraigInterpolant$(D, high(B))$.On pose
– I_1 = ROBDDCraigInterpolant$(D, low(B))$,
– I_2 = ROBDDCraigInterpolant$(D, high(B))$,
En passant aux langages, on a $\mathcal{L}_I = (x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1}) \cap (x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2})$. On note D_0 tel que $\forall v \in \{0,1\}, \mathcal{L}_{v.D_0} = \mathcal{L}_D$, comme la variable x_n n'apparaît pas dans D, le ROBDD de D_0 est le même que celui de D, en considérant qu'on ne travaille plus que sur $n-1$ variables.
Montrons que $\mathcal{L}_D \subseteq \mathcal{L}_I$ et donc que $D \Rightarrow I$:
En prenant la définition d'un interpolant, et le fait que I_1 soit un interpolant de $(D_0, low(B))$, on obtient $\mathcal{L}_{D_0} \subseteq \mathcal{L}_{I_1}$ et donc $x_n.\mathcal{L}_{D_0} \subseteq x_n.\mathcal{L}_{I_1}$ et $\overline{x_n}.\mathcal{L}_{D_0} \subseteq \overline{x_n}.\mathcal{L}_{I_1}$, en passant à l'union, on obtient $(x_n.\mathcal{L}_{D_0} \cup \overline{x_n}.\mathcal{L}_{D_0}) \subseteq (x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1})$. En faissant les mêmes constatations sur I_2, on arrive à $(x_n.\mathcal{L}_{D_0} \cup \overline{x_n}.\mathcal{L}_{D_0}) \subseteq (x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2})$. On peut alors faire l'intersection de ces deux résultats, $(x_n.\mathcal{L}_{D_0} \cup \overline{x_n}.\mathcal{L}_{D_0}) \cap (x_n.\mathcal{L}_{D_0} \cup \overline{x_n}.\mathcal{L}_{D_0}) \subseteq (x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1}) \cap (x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2})$, qui en reprenant la définition de $\mathcal{L}_D$ et de $\mathcal{L}_I$, donne $\mathcal{L}_D \subseteq \mathcal{L}_I$ et donc $D \Rightarrow I$.

Montrons que $\mathcal{L}_B \cap \mathcal{L}_I = \varnothing$ et donc que $B \wedge I = \bot$:
$\mathcal{L}_B \cap \mathcal{L}_I$

$$
\begin{aligned}
&= \mathcal{L}_B \cap \mathcal{L}_B \cap \mathcal{L}_I \\
&= (x_n.\mathcal{L}_{high(B)} \cup \overline{x_n}.\mathcal{L}_{low(B)}) \cap (x_n.\mathcal{L}_{I_1} \cup \overline{x_n}.\mathcal{L}_{I_1}) \\
&\quad \cap (x_n.\mathcal{L}_{high(B)} \cup \overline{x_n}.\mathcal{L}_{low(B)}) \cap (x_n.\mathcal{L}_{I_2} \cup \overline{x_n}.\mathcal{L}_{I_2}) \\
&= \big((x_n.\mathcal{L}_{high(B)} \cap x_n.\mathcal{L}_{I_1}) \cup (x_n.\mathcal{L}_{high(B)} \cap \overline{x_n}.\mathcal{L}_{I_1}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap x_n.\mathcal{L}_{I_1}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap \overline{x_n}.\mathcal{L}_{I_1})\big) \\
&\quad \cap \big((x_n.\mathcal{L}_{high(B)} \cap x_n.\mathcal{L}_{I_2}) \cup (x_n.\mathcal{L}_{high(B)} \cap \overline{x_n}.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap x_n.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap \overline{x_n}.\mathcal{L}_{I_2})\big) \\
&= \big((x_n.\mathcal{L}_{high(B)} \cap x_n.\mathcal{L}_{I_1}) \cup (x_n.\mathcal{L}_{high(B)} \cap \overline{x_n}.\mathcal{L}_{I_1}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap x_n.\mathcal{L}_{I_1})\big) \\
&\quad \cap \big((x_n.\mathcal{L}_{high(B)} \cap \overline{x_n}.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap x_n.\mathcal{L}_{I_2}) \cup (\overline{x_n}.\mathcal{L}_{low(B)} \cap \overline{x_n}.\mathcal{L}_{I_2})\big) \\
&= (x_n.\mathcal{L}_{high(B)} \cap x_n.\mathcal{L}_{I_1}) \cap (\overline{x_n}.\mathcal{L}_{low(B)} \cap \overline{x_n}.\mathcal{L}_{I_2}) \\
&= \varnothing
\end{aligned}
$$

On a donc $B \wedge I = \bot$.

□

En appliquant ce nouvel algorithme 6 sur le calcul d'interpolant de (S_D, S_B), de notre exemple fil rouge, on arrive au ROBDD de la figure 5.6. Si on regarde la taille de ce ROBDD, on constate

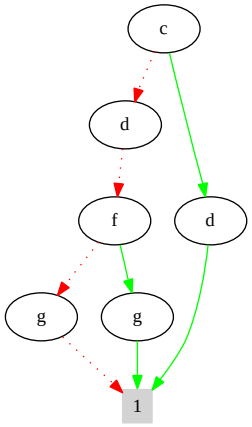


FIGURE 5.6 – Exemple de McMillan 5.2, calcul d'un interpolant par l'algorithme 6.

qu'il ne contient plus que 6 nœuds. Il est donc plus petit que le ROBDD résultant de l'algorithme précédent. Il semble, de plus, être le plus petit ROBDD interpolant. Il représente aussi le prédicat de séparation donné par McMillan.

Chapitre 6

Implémentation

Comme on l'a vu dans le chapitre précédent, on a maintenant des algorithmes pour calculer des interpolants sur les ROBDD. On a donc envie d'évaluer les performances de ces algorithmes.

6.1 Mise en œuvre

Pour mettre en œuvre ces différents algorithmes de calcul d'interpolants, on a utilisé la librairie **BuDDy** [1]. L'avantage de la librairie BuDDy est qu'elle contient toutes les fonctions nécessaires aux manipulations des BDD. Cette librairie utilise, pour la gestion des BDD, un mécanisme de cache. Ce mécanisme permet d'économiser de la mémoire. En effet, le principal problème des BDD est que leur représentation en mémoire est très coûteuse. Pour économiser la mémoire avec ce mécanisme de cache, BuDDy stocke les BDD déjà utilisés et leur attribue une clé de hachage. C'est ensuite cette clé de hachage qui sera utilisée. L'unicité de la clé de hachage implique que deux BDD canoniques ne peuvent être représentés deux fois en mémoire. De plus malgré ce que l'on a dit avant, BuDDy travaille plutôt avec des ROBDD, qui sont réduits toujours grâce au mécanisme de cache.
Comme on vient de le dire, BuDDy présente toutes les fonctions nécessaires aux manipulations de BDD. Grâce à cela, l'implémentation des algorithmes est très naturelle. De plus, l'utilisation du cache permet de tester la valeur des ROBDD, dans notre cas utile pour tester si un ROBDD est réduit à la feuille Vrai ou Faux. D'autre part, BuDDy travaillant sur des ROBDD, la comparaison de variables en fonction de leurs positions dans l'ordre des variables des ROBDD est très simple. Et bien entendu, on trouve dans BuDDy les opérateurs binaires booléens utiles au calcul sur les BDD.
Là où l'utilisation du cache peut poser problème c'est pour son initialisation. En effet, lors de l'initialisation de BuDDy, et de son cache, il faut fixer le nombre de nœuds nécessaires pendant l'utilisation de la libraire. On verra en plus que la modification de la taille du cache nous donne des résultats, non comparables, sur le temps d'exécution des algorithmes. Cela est dû au fait que l'initialisation du cache est une opération très coûteuse en temps.

6.2 Tests et mesures

Pour tester ces différents algorithmes, on a créé automatiquement des ROBDD. Pour cela, et pour chaque génération, on fixe le nombre de variables du ROBDD, ainsi que son nombre maximum de chemins (c'est-à-dire le nombre de mots du langage du ROBDD). La génération va alors consister à choisir aléatoirement le nombre de chemins, inférieur au maximum souhaité, et pour chaque chemin, choisir pour chaque variable si elle doit être vraie ou fausse. Ensuite, les règles de réduction des ROBDD vont pouvoir traiter notre BDD généré et ainsi créer tous les cas

1. http://sourceforge.net/projects/buddy/

particuliers inhérents à ces mêmes règles de réduction. Cependant, les BDD créés ne seront pas forcément (et même sûrement pas) disjoints deux à deux. Il nous faudra alors, avant de calculer un interpolant entre deux ROBDD, retirer l'intersection des deux ROBDD, à un des deux ROBDD. BuDDy produisant automatiquement des ROBDD, cette construction de BDD automatique et automatiquement disjoints, produit forcément des ROBDD. Pour valider les tests, on vérifie aussi que chaque résultat, produit par les algorithmes, soit bien un interpolant.

Ce dispositif de génération automatique de BDD, produit des BDD respectant deux paramètres, le premier fixant le nombre de variables et le second fixant le nombre maximum de chemins du BDD et donc le nombre maximum de nœuds. On va donc pouvoir mesurer l'influence de ces deux paramètres sur le temps et l'utilisation mémoire des différents algorithmes du chapitre précédent. Le temps d'utilisation est mesuré avec la fonction GNU `time`, qu'elle retourne dans une sortie portable, compatible POSIX. Cette sortie contient le temps réel, le temps utilisateur et le temps système de la fonction sur laquelle on fait la mesure. On ne mesure que le temps nécessaire à l'initialisation de BuDDy, au calcul de l'interpolant, et à l'enregistrement de cet interpolant. La consommation mémoire sera elle mesurée directement dans BuDDY. En effet, la consommation de mémoire est équivalente au nombre de nœuds utilisés par BuDDy pour faire ces calculs. Le mécanisme de cache de BuDDy intègre un ramasse-miettes, qu'il a fallu bloquer pour ne pas fausser les mesures.

Dans un premier temps, on fait des mesures sur 40 petits BDD. Ces BDD ont 50 variables, et un maximum de 200 chemins (figure 6.1). Et on fait des mesures sur ces BDD avec les algorithmes 4, 5 et 6, en calculant l'interpolant I entre tous les couples de BDD (D, B). On obtient des résultats relativement bons vis-à-vis du temps puisque en moyenne, le temps nécessaire au calcul d'un interpolant est de 0,2 seconde. Ce résultat est d'autant plus étonnant quant on connaît le temps nécessaire à l'initialisation de BuDDY, qui est aussi de 0,2 seconde. Il faudra donc augmenter la taille des BDD pour pouvoir mesurer réellement le temps mis par l'algorithme. Cependant, on sait que les différentes versions de l'algorithme ont une complexité linéaire par rapport au nombre de noeuds. Et dans notre cas, on a au plus 200 chemins avec 50 variables, donc au plus 10000 noeuds par ROBDD. Ce qui s'énumère assez vite sur une machine moderne. On va alors mesurer la mémoire utilisée, et plus particulièrement le nombre de nœuds utilisés pendant l'algorithme.

	D			B		
	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage
Moyenne	3695,13	99,33	99,33	3695,13	99,33	99,33
Quantile à 0%	282	6	6	282	6	6
Quantile à 25%	1457	36	36	1457	36	36
Quantile à 50%	4097	109	109	4097	109	109
Quantile à 75%	5781	159	159	5781	159	159
Quantile à 100%	6553	180	180	6553	180	180

FIGURE 6.1 – Informations sur D et B, générés pour 50 variables et 200 chemins possibles.

Pour analyser les résultats des figures 6.2, 6.3 et 6.4, on observe dans un premier temps, la différence de nœuds entre D et l'interpolant I. Sur la moyenne des résultats, le nombre de nœuds a été divisé par 40, les résultats sont similaires sur la division moyenne du nombre d'états pour chaque interpolant. On a même eu une division du nombre d'états par plus de 230. Donc pour ce qui est de la réduction de la taille (espace mémoire) de D par l'interpolant I, on ne peut que être convaincu par ces résultats, qui devraient, cependant, être vérifier sur des résultats dans le cadre de l'apporche CEGAR du model-checking. Pour ce qui est du nombre de nœuds utilisés par les

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	105,02	53,68	6533,07	7361,07
Quantile à 0%	14	6	13	688
Quantile à 25%	40	19	485,25	5373,25
Quantile à 50%	98,5	43	2373,5	6834,5
Quantile à 75%	167,5	87	7295,5	9464,25
Quantile à 100%	260	136	63704	13041

FIGURE 6.2 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 4, pour des ROBDD à 50 variables et 200 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	89,42	60,25	7043,35	7492,39
Quantile à 0%	11	6	16	685
Quantile à 25%	37	26	682	5807
Quantile à 50%	87	50	3080	7389
Quantile à 75%	136	99,5	8662	9975
Quantile à 100%	191	136	83648	12981

FIGURE 6.3 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 5, pour des ROBDD à 50 variables et 200 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	89,42	60,25	7043,35	7492,39
Quantile à 0%	11	6	16	685
Quantile à 25%	37	26	682	5807
Quantile à 50%	87	50	3080	7389
Quantile à 75%	136	99,5	8662	9975
Quantile à 100%	191	136	83648	12981

FIGURE 6.4 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 6, pour des ROBDD à 50 variables et 200 chemins maximum possibles.

algorithmes, il est à peu de chose près similaire pour les trois algorithmes, et il est très proche de la somme du nombre de nœuds de D et du nombre de nœuds de B. Sur les données brutes, on arrive à des différences entre le nombre de nœuds utilisés et la somme des nœuds de D et B de l'ordre de 100. Cette faible utilisation de la mémoire est à mettre au bénéfice du cache de BuDDy. On remarque cependant, une petite différence entre le nombre de nœuds utilisés par l'algorithme 4 et le nombre utilisés par l'algorithme 5. Les meilleurs performances de l'algorithme 5 sont dus au fait que dans la règle dite de réduction de variable, on va traiter un cas qui aurait été traité à l'appel récursif suivant dans l'algorithme 4. On a donc sans doute un algorithme plus rapide, mais surtout il consomme un peu moins de mémoire.

Un légère déception peut venir du faite que l'algorithme 6 présente des résultats en tout points similaire aux résultats de 5. Cela est du au fait que la méthode de génération de ROBDD ne crée pas assez de ROBDD ayant des variables manquantes. On va donc augmenter le nombre de chemins possibles à 2000 et réduire le nombre de variables à 10 (avec 10 variables on peut produire 2^10 mots de $\{0,1\}$ soit 1024 chemins). On est dans ce cas sûr d'avoir des cas de simulation de variable.

	D			B		
	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage
Moyenne	177,43	180,94	257,14	202,4	270,85	479,1
Quantile à 0%	51	12	12	147	106	119
Quantile à 25%	143	92,75	108	186,5	215,75	287
Quantile à 50%	181,5	165	208	205	302,5	494
Quantile à 75%	221,25	266,25	377	226	339,5	714
Quantile à 100%	242	374	767	240	370	874

FIGURE 6.5 – Informations sur D et B, générés pour 10 variables et 2000 chemins possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	170,49	168,12	361,82	630,93
Quantile à 0%	56	12	15	267
Quantile à 25%	144	93	158	546,75
Quantile à 50%	174,5	163	312	648
Quantile à 75%	201,25	242,5	561,25	735
Quantile à 100%	234	338	884	850

FIGURE 6.6 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 4, pour des ROBDD à 10 variables et 2000 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	170,97	168,31	372,58	629,14
Quantile à 0%	57	12	16	273
Quantile à 25%	144	93,75	161,75	547
Quantile à 50%	175	163	334	650
Quantile à 75%	201,25	243	573,75	739
Quantile à 100%	235	338	885	791

FIGURE 6.7 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 5, pour des ROBDD à 10 variables et 2000 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	166,84	158,14	375,17	629,6
Quantile à 0%	57	12	17	278
Quantile à 25%	141	88,75	169	549
Quantile à 50%	169	151,5	341	645,5
Quantile à 75%	194	228	572,25	733
Quantile à 100%	234	321	874	799

FIGURE 6.8 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 6, pour des ROBDD à 10 variables et 2000 chemins maximum possibles.

En analysant les figures 6.5, 6.6, 6.7 et 6.8, on observe que le nombre de nœuds de l'interpolant a diminué avec l'utilisation de l'algorithme 6. Les ROBDD produit était bien dans le cadre ou la règle de simulation de variable a été appliqué. On observe donc les résultats de l'interpolant respectant la définition de Craig. Pour l'utilisation mémoire, les résultats de l'algorithme 6 sont un peu moins bon. Cela est du à l'opération $\wedge$ entre les deux "sous-interpolants", qui est plus coûteuse que le $\vee$.

Pour vérifier les résultats obtenus précédemment, on relance une série de générations, toujours pour 50 ROBDD, mais cette fois-ci les ROBDD ont 150 variables et un maximum de 1500 chemins (figure 6.9). On obtient alors les résultats de figures 6.10, 6.11 et 6.12.

	D			B		
	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage
Moyenne	69135	520,4	520,4	69135	520,4	520,4
Quantile à 0%	11475	83	83	11475	83	83
Quantile à 25%	40033	297	297	40033	297	297
Quantile à 50%	71350	536	536	71350	536	536
Quantile à 75%	92891	702	702	92891	702	702
Quantile à 100%	130405	993	993	130405	993	993

FIGURE 6.9 – Informations sur D et B, générés pour 150 variables et 1500 chemins possibles.

On a toujours un nombre de nœuds utilisés proche de la somme des nœuds de D et des nœuds de B. Cette fois-ci l'écart moyen est de 200 nœuds, avec un maximum à 617 nœuds, mais ce qui ne représente une augmentation que de 0,5%. Pour ce qui est des pourcentages de nœuds utilisés par rapport à la somme des nœuds de D et de B, le maximum est de 3,5%, et les plus gros pourcentages apparaissent quand les ROBDD D et B sont les plus petits. Pour le temps nécessaire à l'exécution de l'algorithme, il est dans le cas de ces ROBDD à 150 variables plus conséquent, avec en moyenne 0,35 seconde quelque soit l'algorithme, et sans jamais dépasser la demie seconde.

D'après ces tests, on peut se demander si les résultats de l'algorithme 5 sont meilleurs que ceux de l'algorithme 4. Au niveau du temps d'exécution, les deux algorithmes sont similaires, ceci n'est pas étonnant quant on voit le peu de différences qui les sépare. Pour ce qui est de la consommation mémoire, les performances sont plus à l'avantage de l'algorithme 5, mais sans réellement triompher sur l'autre algorithme. De plus, sur ces exemples générés, la taille du ROBDD interpolant obtenu par l'algorithme 5 est très souvent égale à la taille de celui obtenu par l'algorithme 4, dans certain cas, on observe des différences d'un ou deux nœuds, à l'avantage de l'algorithme 5.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	502,47	327,15	184444,77	131610,54
Quantile à 0%	151	69	1580	39736
Quantile à 25%	389	209	48545	104668
Quantile à 50%	515	342	122855	129309
Quantile à 75%	637	431	258581	159199
Quantile à 100%	883	688	817981	228026

FIGURE 6.10 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 4, pour des ROBDD à 150 variables et 1500 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	425,5	354,37	190171,43	138448,7
Quantile à 0%	118	67	2080	39709
Quantile à 25%	329	222,75	53792	107241,5
Quantile à 50%	442	370	141488	137619
Quantile à 75%	551	468,5	274832	166976
Quantile à 100%	719	707	830208	227879

FIGURE 6.11 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 5, pour des ROBDD à 150 variables et 1500 chemins maximum possibles.

	Nombre de nœuds	Nombre de chemins	Nombre de mots du langage	Nombre de nœuds utilisés
Moyenne	425,5	354,37	190171,43	138448,7
Quantile à 0%	118	67	2080	39709
Quantile à 25%	329	222,75	53792	107241,5
Quantile à 50%	442	370	141488	137619
Quantile à 75%	551	468,5	274832	166976
Quantile à 100%	719	707	830208	227879

FIGURE 6.12 – Mesures sur l'interpolant, I, de (D, B), calculé par l'algorithme 6, pour des ROBDD à 150 variables et 1500 chemins maximum possibles.

Les ROBDD interpolants obtenus sont donc bien toujours plus petits que les ROBDD interpolés, et cela en suivant la relation d'ordre défini dans le chapitre 5, par la définition 11. Le temps d'exécution est quant à lui relativement raisonnable. Pour information, l'algorithme 5 a été testé avec un ROBDD de 200 variables et 900000 nœuds, le ROBDD était construite de telle sorte que l'algorithme nécessite le parcours de tous les nœuds (le préfixe discriminant était pour tous les mots, le dernier symbole). Il a alors fallu moins de 5 secondes pour calculer l'interpolant. Et ce alors que l'initialisation de BuDDy nécessitait déjà 2 secondes. L'occupation mémoire est elle aussi tout à fait raisonnable. D'autant que le parcours étant récursif, le BDD, ne nécessite pas forcément

d'être chargé entièrement. Les résultats globaux de ces algorithmes sont dans la pratique plutôt bon. Ils s'expliquent par la complexité linéaire par rapport au nombre de nœuds du ROBDD à interpoler, et aussi au mécanisme de cache de BuDDy. La complexité linéaire participe à l'exécution rapide des algorithmes. Tandis que le mécanisme de cache de BuDDy participe à l'utilisation raisonnable de la mémoire.

Chapitre 7

Conclusion

Le but de ce mémoire était de trouver une méthode pour séparer les espaces d'états symboliques dans le cadre du raffinement d'une abstraction dans l'approche CEGAR du model-checking. On s'est restreint à la représentation des ensembles d'états par des ROBDD. On a alors défini une méthode pour les séparer, par le calcul d'un interpolant. Pour cela, on a utilisé des résultats sur les langages, et surtout la notion de préfixe discriminant.

On a donc défini la notion de préfixe discriminant, comme les préfixes d'un langage n'appartenant pas à l'autre langage. Et on a vu comment à partir de cette notion, on pouvait définir un interpolant. On crée cet interpolant, simplement en concaténant à chaque mot de l'ensemble des préfixes discriminants toutes les possibilités de mots, pour obtenir un langage dont les mots ont la même taille que ceux des langages interpolés. On a ensuite travaillé sur des structures de données particulières (automates finis à langages dans Σ^n, ROBDD) représentant les langages. A partir des ces structures de données, on a réussi à trouver des algorithmes efficaces, permettant de calculer ces préfixes discriminants et donc l'interpolant que l'on chercher à obtenir. Ces algorithmes se basent sur la structure de représentations de données que l'on a utilisée pour rechercher les préfixes discriminants.

Les résultats obtenus sont conformes à nos attentes. En effet, la nécessité de trouver une partition de l'ensemble des états symboliques d'un système est induite par la nécessité de réduire la taille de la représentation de ces ensembles. Et dans le cas du calcul de notre interpolant, les résultats quant à la réduction de la taille du ROBDD, sont positifs. On arrive à des réductions de taille de l'ordre de 50 pour des ROBDD avec 2000 nœuds et de l'ordre de 150 avec des ROBBD à 60000 nœuds. Cependant, les résultats n'ont été obtenus que sur des ROBDD générés aléatoirement, et que très peu sur des ROBDD obtenus par l'analyse de systèmes.

Les notions nécessaires au calcul d'un interpolant utilisant les préfixes discriminants, sont définies au niveau des langages. On doit donc pouvoir construire des algorithmes de calcul d'interpolants, utilisant ces préfixes discriminants, sur d'autres structures de données représentant des langages.

Travaux futurs

Dans l'analyse de résultats du chapitre 6, il manque l'analyse de l'impact du cache de BuDDy dans le calcul d'un interpolant. On a dit que le fait d'utiliser un cache provoquait des lenteurs d'initialisation, mais on n'a pas cherché à connaître l'impact de l'utilisation du cache sur la construction d'un interpolant. En plus d'évaluer l'impact du cache, il faudrait essayer de l'utiliser. On pourrait imaginer, par exemple, réduire la longueur du parcours de ROBDD en ne parcourant qu'une unique fois un couple de nœuds. On pourrait aussi utiliser la représentation canonique des ROBDD, fa-

cilement récupérable avec le cache, pour réduire encore la taille du ROBDD interpolant.

Pour pouvoir mieux analyser les résultats obtenus dans le cadre d'une approche CEGAR du model-checking, il serait intéressant de comparer ces résultats à d'autres résultats de calcul d'interpolant. Pour cela, il faudrait mettre en place les algorithmes de ce rapport au sein d'un model-checkeur. Les travaux de S. Kiefer et J. Esparza [11] pourraient être intéressants comme base de comparaison. Il faudrait alors intégrer ces nouveaux algorithmes de calcul d'interpolant dans le model-checkeur qu'ils ont utilisé, MOPED [1]. L'inconvénient est que le model-checkeur MOPED n'utilise pas la libraire de gestion de BDD BuDDY utilisé dans ce rapport, mais la libraire CUDD [2]. Avec une telle implémentation, on pourrait observer les différences, dans le cycle CEGAR, induites par l'utilisation d'algorithme de calcul d'interpolant différents.

Les algorithmes décris dans ce rapport, utilisent des structures de données particulières pour calculer les préfixes discriminants. Cependant on fait encore appel aux opérateurs de logiques pour construire un interpolant. On pourrait imaginer utiliser aussi la structure des ROBDD (et automates à langages dans Σ^n pour construire notre interpolant. En effet, si on calcule un interpolant de (D, B), sur des ROBDD, on peut partir de la représentation de D et juste en rajoutant entre un état et les feuilles Vrai et Faux, et en supprimant des arcs, obtenir un interpolant. Il faut noter que les résultats obtenus pour l'exemple fil rouge du chapitre 5, sont calculables par modifications structurelles de l'interpolé (S_D).

La notion de préfixe discriminant étant définie pour les langages de Σ^n, il est envisageable de transposer les algorithmes de calcul d'interpolants à d'autre structures que les automates à langages dans Σ^n et les ROBDD. On peut aussi regarder à lever la restriction sur la comparaison de langage de Σ^n, et voir comment définir les préfixes discriminants dans un cadre plus général. On peut aussi, pour citer quelques structures, imaginer définir de manière similaire, aux ROBDD, des interpolants pour les NDD.

On peut aussi envisager, une parallélisation du calcul d'un interpolant. En effet, le parcours que l'on met en place est très facilement parallélisable. On peut par exemple séparer les appels récursifs (au nombre de deux pour les ROBDD). D'autant plus facilement qu'ils n'utilisent rien de commun, dans le cas général des algorithmes. Il faut cependant imaginer comment partager le cache du gestionnaire de BDD, dans le cas d'une implémentation dans un gestionnaire à cache.

1. http://www.fmi.uni-stuttgart.de/szs/tools/moped/
2. http://vlsi.colorado.edu/~fabio/CUDD/

Bibliographie

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[2] J. R. Burch, E.M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} states and beyond. *Inf. Comput.*, 98(2) :142–170, 1992.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[4] A. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5) :1512–1542, 1994.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterExample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5) :752–794, 2003.

[6] W. Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22(3) :269–285, 1957.

[7] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of 15th Conf. on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[9] H. R. Andersen. An introduction to binary decision diagrams. Lecture notes, http://www.itu.dk/people/hra/bdd97.ps, 1997.

[10] K. L. McMillan. Lazy abstraction with Interpolants. In *Proc. of 18th Conf. on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[11] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In *Proc. of 12th Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2006.