Université Pierre et Marie Curie - Paris VI
Laboratoire d'Informatique de Paris 6 - LIP6

Thèse de Doctorat
Spécialité Informatique

Diagrammes de Décision de Données pour la Vérification de Systèmes Matériels

Vincent Beaudenon

Thèse encadrée par Emmanuelle Encrenaz
Soutenue le 13 décembre 2006

Devant le jury composé de :

M. GREINER Alain	Directeur de thèse
Mme ENCRENAZ Emmanuelle	Co-encadrante de thèse
Mme PETRUCCI Laure	Rapporteur
M. COUVREUR Jean-Michel	Rapporteur
M. KORDON Fabrice	Examinateur
M. BENALYCHERIF Mohamed-Lyes	Examinateur

Je n'ai pas besoin de faire des phrases. J'écris pour tirer au clair certaines circonstances. Se méfier de la littérature. Il faut écrire au courant de la plume ; sans chercher les mots.

Jean-Paul Sartre. *La Nausée*

Combien de marches ai-je descendues avant de pénétrer dans ce bouge ? Sept ? Trente-six ? Il n'est pierre que j'aie effleurée, pas que j'aie accompli, qui ne cachât un hiéroglyphe. Quand je l'aurai dévoilé, à mes féaux sera enfin révélé le Mystère. Après il n'y aura plus qu'à le déchiffrer ; et sa solution sera la Clef, derrière laquelle se cache le Message, qui, à l'initié, et à lui seul, dira en lettres claires quelle est la nature de l'Énigme.

Umberto Eco. *Le Pendule de Foucault* trad. Jean-Noël Schifano.

Avant-Propos

Toute recherche sans dette est une recherche suspecte[1]. En ce sens, les travaux relatés dans ce manuscrit n'ont rien de douteux tant mes créanciers sont nombreux.

Le premier est mon professeur de mathématiques de seconde, M. Soret, qui m'initia à la programmation. Viennent ensuite certains enseignants de l'UPMC et de l'Ecole Centrale de Lyon, M. Samir Akkouche en particulier, qui confirmèrent, peu à peu, mon désir de travailler dans la recherche.

Mes travaux ont été rendus possibles grâce à la confiance réitérée d'Alain Greiner qui m'accueillit dans son équipe dès 1999 et dont les conseils, souvent prodigués à point nommé, m'ont été grandement profitables. Je remercie de même Emmanuelle Encrenaz d'avoir encadré ces travaux, sans elle je n'aurais sans doute jamais entendu l'injonction « Maintenant, tu rédiges. » et le présent document n'existerait pas. Pour leur intérêt accordé à mes travaux, je remercie mes rapporteurs Laure Petrucci (et ses précieuses annotations) ainsi que Jean-Michel Couvreur. Je remercie d'autre part Mohamed-Lyes Benalycherif et Fabrice Kordon d'avoir accepté de faire partie du jury de soutenance. Enfin, je remercie Annick Lablanche de m'avoir aidé à franchir les dernières marches, les plus effrayantes, celles de l'administration.

Je tiens d'autre part, à exprimer ma plus sincère gratitude à l'attention de tous les membres de l'équipe DALI de l'Université de Perpignan. Je conserve d'eux le souvenir d'un accueil exemplaire et d'une équipe remarquablement chaleureuse.

Je remercie également Michelle Guérin pour m'avoir supporté, dans tous les sens du terme, lorsque tout courage m'avait abandonné.

Durant ces longues années de thèse, la collaboration avec les autres doctorants de l'équipe « vérif », Cécile Braunstein, Cédric Roux et Sami Taktak, fut pour moi l'occasion d'échanges fructueux et d'enrichissantes et sympathiques discussions. Mes autres collègues du département ASIM ne sont évidemment pas en reste : ce fut toujours un plaisir de partager avec eux des points de vue d'ordre scientifique ainsi que de bons moments « en terrasse » ou dans les établissements avoisinant le campus de Jussieu. J'adresse à ce sujet une mention spéciale à Christophe Alexandre dont l'éloquence, au sens purement quantitatif, a facilité certains choix techniques, du moins, lorsque la situation ne se prêtait pas à d'autres réjouissances.

Pour finir, et pour leur soutien sans faille, je remercie mes amis et tous les membres de ma famille. C'est en silence qu'ils se mordaient les doigts quand, à leur demande, je leur parlais de mes travaux. Qu'ils soient rassurés : maintenant, c'est fini.

[1] U. Eco, *Comment Voyager avec un Saumon*

Introduction

Ce chapitre présente l'objectif général de notre travail et décrit brièvement le contenu de chaque chapitre.

Objectif de cette Thèse

La vérification formelle d'une propriété ϕ pour un modèle M se résume à démontrer que M *est un modèle* de la propriété ϕ (noté $M \models \phi$). Ce problème se pose de diverses façons, suivant les spécifications de M, de ϕ et des manières de prouver $M \models \phi$. Lorsque M est fini, le *model-checking* rend cette vérification automatisable par une exploration exhaustive des comportements de M, lesquels sont confrontés à ϕ. Cette méthode se heurte généralement à une explosion combinatoire liée à la multiplicité des exécutions *possibles* du modèle. De plus, dans le cas d'applications issues du contexte industriel, afin d'accorder une confiance maximale aux résultats obtenus, il est souhaitable de conserver le modèle fourni par le concepteur ; ceci constitue une source supplémentaire de complexité pour la vérification. Les enjeux de cette thèse portent sur ces deux derniers points :

1. Proposer un outil de vérification par model-checking pour des systèmes décrits dans un langage proche de ceux utilisés par les concepteurs (suffisamment détaillé, avec des primitives de communication complexes).
2. Mettre en œuvre des méthodes de représentation d'ensembles d'états permettant de repousser l'explosion combinatoire.

Nous avons axé notre étude autour de systèmes conçus au sein de l'environnement Disydent [Dis] qui offre un environnement de spécification et de synthèse de systèmes complexes à partir d'une description de haut niveau. Le comportement de ces systèmes est décrit sous la forme d'un ensemble de processus communicant par canaux. Cette sémantique est contenue dans celle du langage Promela sur lequel repose le model-checker SPIN [Hol97b]. Ce model-checker s'appuie sur une représentation explicite d'espace d'états pour vérifier des propriétés LTL. Cette méthode a fait ses preuves pour la vérification de programmes C et de protocoles [Hol91], mais elle est moins efficace que les méthodes symboliques pour des systèmes fortement indéterministes ou parallèles [PTK03].

Notre objectif est de mettre en œuvre des méthodes symboliques de vérification de propriétés CTL sur des systèmes écrits dans un sous-ensemble du langage Promela suffisant pour contenir DPN, en nous basant sur les Diagrammes de Décision de Données (DDD, [CEPA$^+$02]). Il s'agit de Diagrammes Orientés Acycliques (DAG) munis d'opérateurs inductifs qui les distinguent des Diagrammes de Décision Binaires (BDD, [Bry86]) et des nombreuses représentations qui en sont dérivées [Bry95].

Nous avons d'abord défini les briques de bases nécessaires à la réalisation d'un model-checker basé sur les DDD. L'expérimentation a mis en évidence les avantages des DDD pour l'analyse de systèmes parallèles et fait ressortir des points à améliorer (par rapport à d'autres outils).

Nous avons ensuite exploité les premières conclusions tirées ainsi que sur les récentes évolutions constatées que constituent les diagrammes de Décision d'Ensembles (SDD, [TM04, CTM05]) et les techniques dites « de saturation » [CC04]. Ceci nous a mené à l'élaboration d'un outil de vérification symbolique plus abouti, faisant intervenir des méthodes de génération d'espaces d'états plus efficaces qui tirent parti de la hiérarchie des systèmes étudiés.

Ce second prototype est utilisé pour vérifier des propriétés sur des systèmes issus du contexte industriel. Les expérimentations corroborent les conclusions formulées au terme des deux précédentes étapes : Le model-checker symbolique à base de DDD hiérarchique permet de vérifier automatiquement des propriétés de systèmes « Disydent » de moyenne complexité. Si les systèmes présentent un fort degré de concurrence, ce model-checker est bien plus performant que les techniques énumératives de SPIN et que les techniques symboliques basées sur les BDD. Enfin, nous proposons des voies de recherche à court et à moyen terme pour la vérification effective de systèmes matériels.

Organisation du document

Première Partie : Systèmes DPN, Model-Checking et Diagrammes de Décision de Données

La première partie de ce document est consacrée à un état de l'art des méthodes de model-checking, de représentation et de génération d'espaces d'états, ainsi qu'à une présentation du sous-ensemble de Promela choisi.

Chapitre 1 : Problématique

Ce chapitre décrit la problématique de cette thèse. Nous présentons d'abord l'environnement Disydent [Dis] et la description DPN (Disydent Process Network). La sémantique DPN est très voisine de celle du langage Promela pour lequel il existe un model-checker fortement répandu, SPIN, qui permet la vérification de propriétés de logique temporelle linéaire (LTL, [Pnu77]). Nous décrivons le fonctionnement de SPIN ainsi que l'étude réalisée par Hong Peng *et al.* [PTK03] qui le compare aux méthodes symboliques de model-checking.

Nous proposons l'emploi de méthodes symboliques pour des systèmes écrits en Promela. Le choix des méthodes symboliques est justifié par leur meilleure capacité à appréhender des systèmes à forte concurrence. Nous devons décrire les mécanismes mis en œuvre par ces méthodes. Le choix de Promela permet de disposer d'un cadre pertinent de comparaison avec les techniques existantes de model-checking. Nous devons en extraire un sous-ensemble suffisant pour simuler les applications DPN.

Chapitre 2 : Model-Checking Symbolique

Ce chapitre décrit la logique CTL [CES86] et les algorithmes de model-checking symbolique de propriétés CTL. Nous décrivons ensuite les principales représentations symboliques d'ensemble d'états :

- les BDD [Bry86] et leurs dérivés [Bry95],
- les Diagrammes de Décision Multivalués (MDD, [CLS99]),

– et les Diagrammes de Décision de Données (DDD, [CEPA+02]) sur lesquels repose cette thèse.
Les DDD sont les plus adaptés pour le type de systèmes que nous souhaitons vérifier. Ce chapitre fait ressortir l'absence d'un cadre formel pour garantir aux DDD des propriétés indispensables au model-checking (canonicité et « bonne définition »). Enfin, il recense l'ensemble des opérateurs sur DDD que nous devrons réaliser pour la vérification de propriétés CTL (les opérateurs $select$, pre et $post$).

Chapitre 3 : Choix d'un sous-ensemble de Promela

Ce chapitre définit un sous-ensemble du langage Promela suffisant pour modéliser des systèmes respectant la sémantique DPN. Ce sous-ensemble permet la modélisation d'un réseau statique de processus concurrents asynchrones qui communiquent au travers de canaux ayant une politique d'accès de type FIFO.

Deuxième Partie : Développement - Contributions

Cette partie décrit les contributions, formelles et techniques, apportées par cette thèse.

Chapitre 4 : DDD ordonnés

Ce bref chapitre décrit un cadre formel de construction des DDD pour garantir leur canonicité et leur « bonne définition ». Ce cadre est basé sur une notion d'*ordre* adaptée aux méthodes inductives et aux affectations multiples.

Chapitre 5 : Approche Promela-DDD

Ce chapitre montre comment construire un model-checker symbolique pour le sous-ensemble de Promela défini au chapitre 3. Dans un premier temps, nous montrons comment réaliser l'opérateur $select$ ainsi que les opérateurs $post$ et pre pour les seules instructions d'affectation d'expressions arithmétiques, ou de lecture-écriture dans des canaux. Nous avons implémenté un premier prototype et nous avons comparé ses performances par rapport à SPIN et aux BDD. Il en ressort que les DDD sont une structure de données efficace pour vérifier des systèmes écrits dans le sous-ensemble de Promela que nous avons extrait. Néanmoins, les DDD perdent en efficacité s'ils comportent de longs chemins, ce qui se produit lorsque le système comprend un grand nombre de variables de type structuré.

Chapitre 6 : DDD hiérarchiques

Ce bref chapitre présente les Diagrammes de Décision d'Ensembles (SDD, apparus en 2004/2005, [TM04, CTM05]), introduisant la hiérarchie de DDD et résolvant les problèmes de longueur des chemins évoqués au chapitre 5. Nous démontrons formellement que les SDD sont des DDD malgré leur différence d'implémentation.

Chapitre 7 : Approche Promela-SDD

Ce chapitre définit la construction d'un état de programme Promela en DDD hiérarchique et précise les définitions des homomorphismes dans le cadre hiérarchique des DDD. Les expérimentations démontrent les apports décisifs de l'approche hiérarchique ainsi que des méthodes de saturation.

Troisième Partie : Applications, Conclusion et Perspectives

Cette partie présente les tentatives de vérification de propriétés sur des systèmes réels avec l'outil de model-checking symbolique basé sur les DDD hiérarchiques. Les résultats montrent effectivement l'avancée apportée par notre technique pour des systèmes présentant un fort degré de concurrence. Nous apportons une conclusion sur notre approche et proposons des perspectives de recherche à court et à moyen terme.

Chapitre 8 : Exemples réalistes

Dans ce chapitre nous appliquons nos méthodes sur des systèmes qui correspondent à une spécification de modèle réellement implémentée dans un but non académique. Il s'agit de deux protocoles (ZCSP [GDDW02] et ANI [Tak03]), qui ont un comportement fortement séquentiel, mais n'ont pu être analysés avec SPIN de façon satisfaisante. Le troisième système est un réseau d'interconnections étagé (également appelé SPIN) dont le comportement très concurrent se prête mieux à l'analyse par des model-checkers symboliques. Ces résultats corroborent les observations faites aux chapitres 5 et 7 : Les techniques énumératives utilisées par SPIN sont les plus efficaces pour les systèmes présentant un faible degré de concurrence. La tendance s'inversant très nettement en faveur du model-checker symbolique à base de DDD hiérarchiques à mesure que le degré de concurrence s'accroît.

Chapitre 9 - Conclusion et Perspectives

Dans ce chapitre, nous dressons un bilan sur notre approche, par rapport aux objectifs que nous nous sommes fixés. Nous recensons les éléments perfectibles des techniques mises en œuvre et proposons des axes de recherche à court et à moyen terme. Le premier tire parti de l'expérience accumulée sur l'utilisation de la bibliothèque [DDD] afin d'améliorer cette dernière. Le second propose d'inclure la définition hiérarchique d'applications DPN en vue de construire un model-checker qui puisse être intégré dans la chaîne Disydent.

Table des matières

Première partie

Systèmes DPN, Model-Checking et Diagrammes de Décision de Données

Chapitre 1

Problématique

Ce chapitre pose quatre questions : Pour quels systèmes souhaitons-nous appliquer des techniques de model-checking ? Quelles sont les techniques existantes ? Sont-elles suffisantes ? Dans le cas contraire, quelles sont les voies susceptibles d'être explorées ? Les réponses que nous avons apportées ont conditionné l'axe principal de recherche que nous exposons dans ce manuscrit.

Sommaire

1.1 Systèmes multiprocesseurs intégrés sur puce

La complexité croissante des systèmes informatiques pose la question de la mise en œuvre de méthodes automatiques pour leur vérification formelle. Par système, on entend généralement un ensemble de composants logiciels qui s'exécutent sur un matériel prédéfini. Cette différence entre le matériel et le logiciel s'estompe. Certaines tâches critiques sont directement réalisées par des composants matériels et la conception de système multiprocesseurs intégrés sur puce incite à considérer le système comme une application unique sans préjuger du caractère d'implantation de ses composants.

Les travaux de Denis Hommais, Frédéric Pétrot et Ivan Augé [HPA01] proposent l'utilisation du formalisme des réseaux de Kahn [Kah74], pour représenter le parallélisme « gros grain » de ces applications. Cette approche, adaptée au contexte industriel [dSv$^+$00], a donné naissance à la chaîne de développement Disydent (pour DIgital System Design ENvironmenT [Dis]). Cette chaîne fournit un cadre de conception de systèmes embarqués dont certaines parties sont implantées en matériel et d'autres en logiciel. Elle est basée sur le formalisme DPN (pour Disydent Process Network) qui décrit un ensemble de processus qui communiquent au travers de canaux de communication ayant une politique d'accès de type FIFO.

Par ailleurs, d'importantes recherches sont en cours pour la vérification automatique de systèmes écrits en langage Promela (pour PROcess MEta LAguage). Il s'agit d'un langage de description d'ensembles de processus asynchrones qui communiquent à l'aide de variables partagées ou de canaux synchrones ou bufferisés (suivant, par exemple, la politique FIFO). Il est possible de vérifier des propriétés temporelles sur ces systèmes à l'aide du model-checker SPIN (pour Simple Promela INterpreter, [Hol97b]).

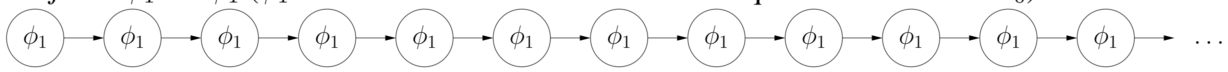
Les canaux de communication point-à-point de DPN et les canaux FIFO de Promela ont des sémantiques très voisines (voir chapitre 3). Plus précisément, la sémantique de Promela contient celle de DPN. Le langage Promela étant largement utilisé dans le domaine du model-checking, nous avons choisi d'orienter cette thèse autour de celui-ci, tout en nous restreignant à DPN.

1.2 Vérification de propriétés LTL avec SPIN

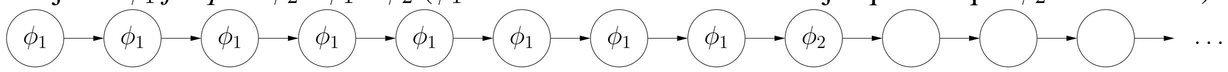
1.2.1 La logique temporelle linéaire LTL

La logique temporelle linéaire (LTL [Pnu77]) permet de spécifier des propriétés le long des chemins d'exécution d'un système de transitions. Ces propriétés dynamiques font référence à un temps discret. Elles s'expriment à l'aide d'opérateurs « temporels » spécifiques résumés dans la figure 1.1 :

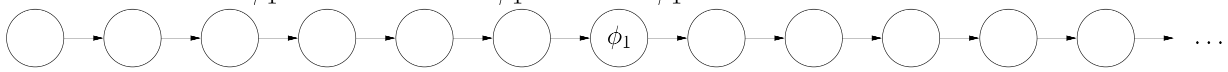
- *Toujours* ϕ_1 : $\mathbf{G}\phi_1$ (ϕ_1 est vraie dans *tous* les états de la séquence débutant en s_0)



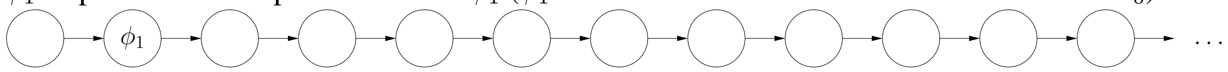
- Toujours ϕ_1 *jusqu'à* ϕ_2 : $\phi_1\mathbf{U}\phi_2$ (ϕ_1 est vraie dans tous les états jusqu'à ce que ϕ_2 le devienne)



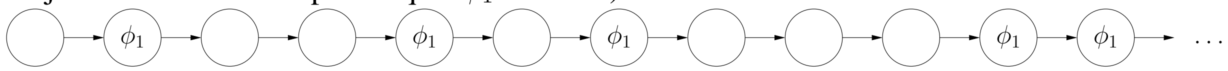
- Au moins une fois ϕ_1 dans le futur : $\mathbf{F}\phi_1 = vrai\mathbf{U}\phi_1$



- ϕ_1 se produira à l'étape *suivante* : $\mathbf{X}\phi_1$ (ϕ_1 est vraie dans l'état suivant immédiatement s_0)



- ϕ_1 se produira *infiniment souvent* : $\mathbf{GF}\phi_1$ (aussi loin que l'on aille dans la séquence, on pourra toujours trouver un état pour lequel ϕ_1 est vraie)



- Dans le futur, ϕ_1 se produira toujours : $\mathbf{FG}\phi_1$ (à partir d'un certain rang, ϕ_1 sera toujours vraie)

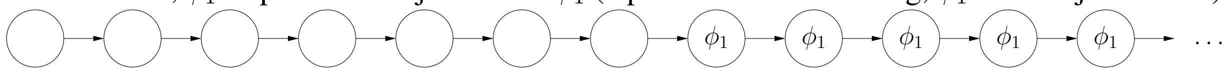


FIG. 1.1 Opérateurs LTL[Pnu77]

Un programme est représenté par un système de transitions fini. Chaque séquence d'exécution est représentée par une succession d'états (représentés par des ronds) dont l'enchaînement est conditionné par les transitions du système (représentées par des flèches). Chaque état représente une configuration de valeurs des variables du système. Si la configuration d'un état satisfait une propriété, cette dernière est inscrite dans le rond représentant l'état.

En pratique, les propriétés vérifiées font intervenir plusieurs opérateurs temporels ($\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$ ou $\mathbf{U}$) composés entre eux. Par exemple, la formule $\mathbf{F}\phi_1$ est vraie pour un état donné si toutes les séquences issues de cet état contiennent au moins un état qui vérifie ϕ_1, et $\mathbf{G}\phi_2$ est vraie si *tous* les états de *toutes* les séquences vérifient ϕ_2. Dans ce cas, si $\phi_2 = \mathbf{F}\phi_1$, alors $\mathbf{G}\phi_2 = \mathbf{GF}\phi_1$ est validée si *pour tous* les états du chemin, la propriété ϕ_1 sera vérifiée au moins une fois dans le futur. On dit alors que ϕ_1 est vérifiée infiniment souvent. Cette propriété permet de vérifier la vivacité d'un élément d'un système à partir d'une formule atomique indiquant son activation (ici ϕ_1). On l'utilise généralement pour déceler des interblocages dans des systèmes concurrents. De la même façon, la composition $\mathbf{FG}\phi_1$ exprime le fait que ϕ_1 soit systématiquement satisfaite à partir d'un certain moment.

Automates de Büchi

La méthode la plus répandue de vérification de propriétés LTL est basée sur la construction du produit synchronisé du système à vérifier et d'un automate (appelé *automate de Büchi*) permettant de reconnaitre des séquences infinies validant une propriété LTL [VW86]. Un automate de Büchi est un automate fini d'états, dont certains sont appelés « états d'acceptation ». Une séquence d'un automate de Büchi est reconnue (ou acceptée) si elle comporte un cycle contenant au moins un état d'acceptation. A partir de la négation de la formule LTL à vérifier, on construit l'automate de Büchi permettant de reconnaître un contre-exemple à la propriété. Le processus de vérification de propriétés LTL est donné sur la figure 1.2. Il commence par l'interprétation du modèle M sous la forme d'un automate fini d'états et de transitions A_M. Ensuite on procède à la construction de l'automate de Büchi $A_{\neg\phi}$ permettant de reconnaître un contre-exemple de la propriété ϕ. Puis on effectue le produit synchronisé $A_M \otimes A_{\neg\phi}$ qui est aussi un automate de Büchi. Deux possibilités s'offrent alors à nous, suivant l'absence ou la présence de cycle d'acceptation (*i.e.* de contre-exemple) dans $A_M \otimes A_{\neg\phi}$:

- soit il n'existe pas de cycle d'acceptation dans $A_M \otimes A_{\neg\phi}$, alors $M \models \phi$,
- soit il existe au moins un cycle d'acceptation. On peut alors analyser les contre-exemples en observant la suite des états de A_M atteints dans chaque cycle d'acceptation. Le diagnostic doit mener à la correction du modèle M, ou à une modification de la propriété ϕ afin d'écarter les scenarii aberrants.

Dans tous les cas, la construction automatique de $A_M \otimes A_{\neg\phi}$ constitue l'étape la plus consommatrice en temps de calcul et en espace mémoire. En terme de complexité, [SC85] a montré que la vérification de propriétés LTL est un problème PSPACE-complet avec une complexité temporelle en $2^{O(|\phi|)} \times O(|M|)$, où $|\phi|$ désigne le nombre maximal d'opérateurs LTL imbriqués dans ϕ, et $|M|$ le nombre d'états accessibles du système.

Vérification *à la volée*

S'il est nécessaire de construire l'automate produit dans sa totalité pour prouver que $M \models \phi$, la réfutation de cette propriété peut être prouvée par l'existence d'un unique contre-exemple. La méthode de vérification dite « *à la volée* » consiste à rechercher le cycle d'acceptation en même temps que l'on construit $A_M \otimes A_{\neg\phi}$ et à interrompre sa construction au premier cycle d'acceptation trouvé [GPVW95]. La construction complète de $A_M \otimes A_{\neg\phi}$ n'est plus nécessaire puisque l'assertion

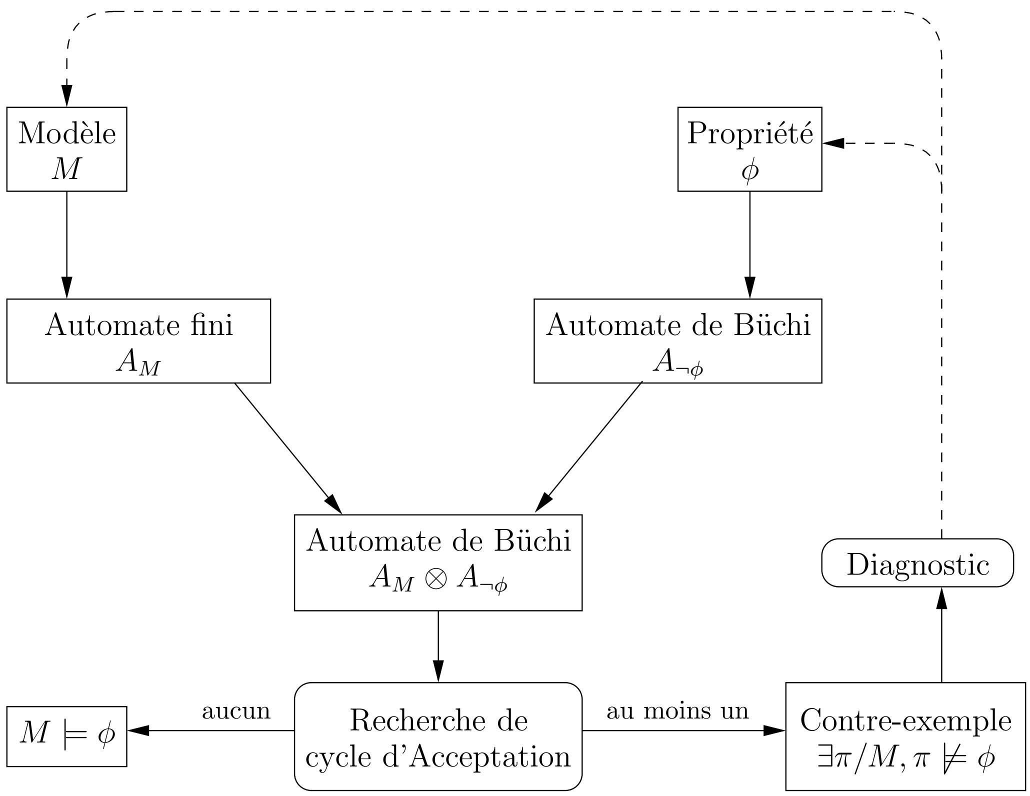


FIG. 1.2 Cycle de vérification de propriétés LTL

$M \models \phi$ est fausse. La méthode de vérification à la volée procède à une construction *en profondeur d'abord* du graphe d'exécution de l'automate produit.

1.2.2 SPIN

SPIN est un outil de vérification de propriétés LTL pour des systèmes distribués, développé par une équipe dirigée par Gerard J. Holzmann des laboratoires Bell, qui a permis la vérification effective de protocoles de communication et de programmes C [Hol97b].

SPIN est en fait un interpréteur de systèmes décrits en langage Promela que nous détaillerons au chapitre 3. Cet interpréteur, propose une *simulation* aléatoire ou interactive du système, ainsi que la génération d'un code C optimisé pour la vérification à la volée de propriétés LTL. L'algorithme de recherche de cycle est une version optimisée [GH93] de l'algorithme de Tarjan pour la recherche de cycles [Tar72] pour extraire une séquence invalidant la propriété à vérifier. Ce contre-exemple pourra être analysé par le module de simulation de SPIN. Ainsi, toutes les étapes du cycle de la figure 1.2 peuvent être réalisées à l'aide de SPIN. La recherche de cycles d'acceptations se fait *en profondeur d'abord*, et utilise une représentation explicite des états.

Cette technique étant particulièrement consommatrice en ressources matérielles, SPIN a été pourvu de nombreuses améliorations. Citons, entre autres, la limitation de l'espace de recherche par réduction d'ordre partiel [HP94, Pel94] et la diminution de l'espace mémoire nécessaire par compression de l'espace d'états [Hol97a] ou par table de hachage [Hol98]. Néanmoins, SPIN reste limité à l'analyse de systèmes représentables sur 10^6 à 10^7 états au maximum. De plus, comme pour chaque outil de model-checking, l'appréhension de systèmes réels est conditionnée par les connaissances de l'utilisateur quant aux méthodes mises en œuvre par SPIN.

1.3 Autres techniques

Les méthodes décrites dans la section 1.2 font mention d'une représentation explicite d'états pour la recherche en profondeur d'abord de cycle sur un graphe. Le principe des méthodes symboliques est de manipuler les états par *ensembles* plutôt qu'individuellement. Nous décrivons ici les principaux travaux de recherche effectués pour la vérification symbolique de systèmes, puis nous explicitons le lien entre ce type d'approche et notre problématique.

1.3.1 Quelques model-checkers symboliques

Le premier model-checker symbolique est apparu grâce aux travaux de J. C. Madre, O. Coudert, M. Currat, A. Debreil, et C. Berthet pour la chaîne SIAM-PRIAM des laboratoires Bull [CBM90b, MCC$^+$90] qui ont donné l'outil FormalCheck. Concurremment, le model-checker SMV (Symbolic Model Verifier [SMV]) a été développé par l'équipe de l'Université de Carnegie Mellon dirigée par Edmund Clarke. Il reprend les principes de l'article « *Symbolic Model Checking :* 10^{20} *states and Beyond* » [BCM$^+$90]. Cet article préconise l'emploi des Diagrammes de Décision Binaires (BDD, [Bry86]) pour une représentation compacte d'ensemble d'états.

Depuis 1999, SMV est remplacé par NuSMV [CCG$^+$02] qui associe la vérification symbolique utilisant une représentation à base de BDD aux méthodes de résolution de problèmes SAT, et à Cadence-SMV pour les raisonnements par abstraction et composition [Cad].

Le mode de définition du modèle à vérifier dépend souvent du milieu d'application du model-checker. Dans le monde académique, des représentations abstraites (comme les réseaux de Petri) sont préférées pour leur concision. Dans le monde industriel, on emploie des représentations proches (voire identiques) de l'implémentation réelle. A l'instar de SPIN, l'outil de vérification symbolique, VIS (Verification Interacting with Synthesis [BHSV$^+$96]), basé sur les BDD, est l'un des rares model-checkers « académiques » à prendre comme point d'entrée des modèles proches des applications industrielles (VERILOG).

Enfin, l'outil SMART (pour Stochastic Model checking Analyzer for Reliability and Timing [SMA]), développé par l'équipe de l'université de Californie (Riverside) dirigée par Gianfranco Ciardo, est basé sur les Diagrammes de Décision Multivalués [CLS99] et les techniques dites « de saturation » [CC04]. Les performances de SMART pour l'analyse de réseaux de Petri ouvrent des perspectives de recherche prometteuses pour le model-checking symbolique.

1.3.2 VIS vs. SPIN

Nos travaux préliminaires ont porté sur la vérification de systèmes à l'aide de SPIN [BED03] : Nous avons démontré des propriétés de vivacité et de terminaison pour le protocole ZCSP [GDDW02] mais au prix d'une simplification extrême du système, ce qui réduit la confiance que l'on peut leur accorder pour le modèle implémenté. De plus, les modifications effectuées pour permettre à SPIN d'achever les calculs nécessitaient une connaissance approfondie du model-checker. Ces carences nous ont mené à identifier d'autres voies de recherche pour vérifier des systèmes matériels complexes à partir d'une description précise.

La comparaison entre les outils SPIN et VIS [PTK03] pour le protocole ATMR donne des éléments de réponse à ce problème. Sur des modèles déduits de l'implémentation fournie par le concepteur, cet article décrit l'application de deux méthodes différentes de vérification et souligne leurs points forts. Il ressort de cette étude une indéniable efficacité de SPIN par rapport à VIS sur des systèmes de petite taille, la tendance s'inversant avec l'augmentation du nombre de processus parallèles. Or, ces deux outils sont fondés sur deux paradigmes antagonistes du model-checking :

- SPIN vérifie des propriétés LTL, s'appuie sur une représentation explicite de l'espace d'états et prend en entrée un réseaux de processus concurrents asynchrones.
- VIS vérifie des propriétés respectant une autre logique temporelle, appelée CTL (que nous détaillerons au chapitre 2). Il s'appuie sur une représentation symbolique de l'espace d'états et est basé sur des modèles *synchrones*.

Aussi, VIS n'est pas non plus adapté pour vérifier des propriétés temporelles sur des systèmes respectant la sémantique de DPN puisque ces derniers définissent des ensembles de processus concurrents *asynchrones*. Cependant, les méthodes symboliques, utilisées par VIS, apparaissent plus adaptées pour appréhender le parallélisme. Il serait donc souhaitable d'appliquer les techniques de model-checking symbolique sur des systèmes décrits en Promela.

1.4 Conclusion

Nous souhaitons appliquer des techniques de model-checking sur des ensembles de processus concurrent et asynchrones communiquant par canaux. De tels systèmes peuvent être écrits en langage Promela, pour lequel il existe un model-checker très répandu, SPIN, qui procède à la vérification à la volée de formules LTL. Cet outil est efficace pour des systèmes séquentiels mais il ne peut appréhender de système massivement parallèle, contrairement aux outils s'appuyant sur les méthodes de vérification symbolique. Aussi, l'emploi de ces méthodes nous est apparu mieux adapté pour la vérification de certains systèmes intégrés sur puce.

Afin de disposer d'un cadre de comparaison pertinent avec les techniques de model-checking existantes, nous avons choisi d'adopter Promela comme langage de modélisation. Dans ce contexte, nous devons déterminer quelle technique de vérification symbolique semble la mieux adaptée. Notre objectif n'étant pas d'implanter Promela mais de simuler des applications DPN, nous devons aussi extraire un sous-ensemble nécessaire et suffisant pour DPN, puis en dériver une représentation formelle dans le but de vérifier des propriétés logico-temporelles.

Chapitre 2

Model-checking Symbolique

Ce chapitre définit les principes utiles à la mise en œuvre des méthodes de vérification symbolique et présente les principaux types de données abstraits utilisés. Nous décrivons le cadre formel de la vérification symbolique de propriétés CTL. Celui-ci est basé sur une représentation compacte d'*ensembles* d'états et de transitions, dont la plus connue utilise les Diagrammes de Décision Binaires. Nous présentons les structures qui en dérivent, plus particulièrement les MDD et les DDD.

Sommaire

2.1 Vérification symbolique de propriétés CTL

Dans cette section nous décrivons la logique temporelle CTL ainsi que les principes généralement utilisés pour leur vérification symbolique.

2.1.1 La Logique Temporelle Arborescente CTL

A l'instar de la logique LTL (section 1.2.1), la logique CTL s'applique sur des modèles décrits à l'aide d'une structure de Kripke définie comme suit :

Définition 2.1 (Structure de Kripke [CGP99]) *Une structure de Kripke est un tuple* $M = (AP, S, L, R, S_0)$ *où*

- AP *est un ensemble fini de propositions atomiques.*
- S *est un ensemble fini d'états.*
- $L : S \to 2^{AP}$ *est une fonction associant à chaque état un ensemble de propositions atomiques vraies dans cet état.*
- $R \subseteq S \times S$ *est une relation de transition. Celle-ci doit être totale* ($\forall s \in S, \exists s'/(s, s') \in R$)*.*
- $S_0 \subseteq S$ *est l'ensemble des états initiaux.*

On appelle *chemin* une séquence infinie d'états $\pi = \pi_0, \pi_1, \pi_2 \ldots$ tels que $\forall i \geq 0$, $(\pi_i, \pi_{i+1}) \in R$. Un état s est dit *accessible* à partir de S_0 s'il existe un chemin $\pi = \pi_0, \pi_1, \pi_2 \ldots$ tel que $(\pi_0 \in S_0) \wedge (\exists i, \pi_i = s)$. La logique CTL est munie des opérateurs traditionnels de la logique booléenne, et de huit opérateurs temporels quantifiés : $\mathbf{EX}$, $\mathbf{AX}$, $\mathbf{EU}$, $\mathbf{AU}$, $\mathbf{EF}$, $\mathbf{AF}$, $\mathbf{EG}$ et $\mathbf{AG}$.

Définition 2.2 (Formule CTL [CES86]) *Les formules CTL sont construites à partir des propositions atomiques* AP *d'une structure de Kripke. Leur syntaxe et leur sémantique sont les suivantes* (ϕ_1 *et* ϕ_2 *sont des formules CTL,* ap *est une proposition atomique) :*

$$
\begin{array}{rcl}
M, s \models ap & \iff & ap \in L(s) \\
M, s \models \neg\phi_1 & \iff & M, s \not\models \phi_1 \\
M, s \models \phi_1 \wedge \phi_2 & \iff & (M, s \models \phi_1) \wedge (M, s \models \phi_2) \\
M, s \models \phi_1 \vee \phi_2 & \iff & (M, s \models \phi_1) \vee (M, s \models \phi_2) \\
M, s \models \mathbf{EX}\phi_1 & \iff & \exists \pi, \pi_0 = s \implies M, \pi_1 \models \phi_1 \\
M, s \models \mathbf{EF}\phi_1 & \iff & \exists \pi, \exists i \geq 0, \pi_0 = s \implies M, \pi_i \models \phi_1 \\
M, s \models \mathbf{EG}\phi_1 & \iff & \exists \pi, \forall i \geq 0, \pi_0 = s \implies M, \pi_i \models \phi_1 \\
M, s \models \mathbf{E}\phi_1\mathbf{U}\phi_2 & \iff & \exists \pi, \exists k \geq 0, \pi_0 = s \implies (M, \pi_k \models \phi_2) \wedge (\forall 0 \leq i < k, (M, \pi_i \models \phi_1)) \\
M, s \models \mathbf{AX}\phi_1 & \iff & \forall \pi, \pi_0 = s \implies M, \pi_1 \models \phi_1 \\
M, s \models \mathbf{AF}\phi_1 & \iff & \forall \pi, \exists i \geq 0, \pi_0 = s \implies M, \pi_i \models \phi_1 \\
M, s \models \mathbf{AG}\phi_1 & \iff & \forall \pi, \forall i \geq 0, \pi_0 = s \implies M, \pi_i \models \phi_1 \\
M, s \models \mathbf{A}\phi_1\mathbf{U}\phi_2 & \iff & \forall \pi, \exists k \geq 0, \pi_0 = s \implies (M, \pi_k \models \phi_2) \wedge (\forall 0 \leq i < k, (M, \pi_i \models \phi_1))
\end{array}
$$

La logique CTL est une logique d'état. Pour un état s donné, le quantificateur de chemin $\mathbf{E}$ indique qu'il existe *au moins* un chemin issu de s, le long duquel une propriété est vraie, et le quantificateur de chemin $\mathbf{A}$ exprime que *tous* les chemins issus de s satisfont cette même propriété. En terme de complexité, [AC88] et [CES86] ont montré que la vérification de propriétés CTL est un problème PSPACE-complet avec une complexité temporelle en $O(|M| \times |\phi|)$ ($|M|$ est le nombre d'états de M et $|\phi|$ le nombre d'opérateurs imbriqués dans ϕ).

2.1.2 Opérateurs nécessaires pour la vérification symbolique

La vérification symbolique de propriétés CTL utilise un type de données abstrait pour la représentation d'ensembles d'états. Il a été démontré [EC80] que ce type de données abstrait devait être muni des opérateurs ensemblistes traditionnels (union, intersection et différence) ainsi que de trois opérateurs suivants :

1. La sélection ($select$) dans l'ensemble $S_1 \subseteq S$ du sous-ensemble des états satisfaisant une condition booléenne ϕ :
$$S_2 = select_\phi(S_1) = \{s_2 \in S_1, (M, s_2 \models \phi)\}$$
2. Le calcul ($post$) de l'ensemble S_2 des états successeurs à partir d'un ensemble d'états $S_1 \subseteq S$ par la relation de transition R :
$$S_2 = post(S_1) = \{s_2, \forall(s_1, s_2) \in S_1 \times S, (s_1, s_2) \in R\}$$
3. Le calcul (pre) de l'ensemble S_1 des états prédécesseurs à partir d'un ensemble d'états $S_2 \subseteq S$ par la relation de transition R :
$$S_1 = pre(S_2) = \{s_1, \forall(s_1, s_2) \in S \times S_2, (s_1, s_2) \in R\}$$

Ces trois opérateurs sont définis à partir de la structure de Kripke sur laquelle on souhaite vérifier des propriétés. Le cadre de la figure 2.1 définit ces trois opérateurs et montre comment vérifier n'importe quelle propriété CTL à l'aide de ces derniers. L'ensemble $Reached$ est obtenu par application itérée de $post$ sur S_0, puis sur $post(S_0)$, l'opération est répétée jusqu'à l'obtention d'un point fixe $\mu Q[S_0 \cup post(Q)] = S_0 \cup post(S_0 \cup post(\ldots))$.

On note sat_ϕ, l'ensemble des états de $Reached$ qui satisfont la propriété CTL ϕ.

Opérateurs pre et $post$, états accessibles :
$$S_2 = post(S_1) = \{s_2, \forall(s_1, s_2) \in S_1 \times S, (s_1, s_2) \in R\}$$
$$S_1 = pre(S_2) = \{s_1, \forall(s_1, s_2) \in S \times S,_2 (s_1, s_2) \in R\}$$

Accessibilité :
$$s_1 \in Reached, \forall s_2, (s_1, s_2) \in R, s_2 \in Reached$$
$$Reached = S_0 \cup post(S_0) \cup post(post(S_0)) \cup \ldots$$
$$Reached = \mu Q[S_0 \cup post(Q)]$$

Propositions atomiques :
$$M, s \models \phi_1 \Leftrightarrow s \in select_{\phi_1}(Reached)$$

Opérateurs Booléens :
$$M, s \models \neg\phi_1 \Leftrightarrow s \in Reached \backslash sat_{\phi_1}$$
$$M, s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \in sat_{\phi_1} \cap sat_{\phi_2}$$
$$M, s \models \phi_1 \vee \phi_2 \Leftrightarrow s \in sat_{\phi_1} \cup sat_{\phi_2}$$

Opérateur $\mathbf{EX}$:
$$M, s \models \mathbf{EX}\phi_1 \Leftrightarrow s \in pre(sat_{\phi_1}(Reached))$$

Equivalences CTL :
$$\begin{aligned} \mathbf{AX}\phi_1 &= \neg\mathbf{EX}(\neg\phi_1) \\ \mathbf{EF}\phi_1 &= \mathbf{E}true\mathbf{U}\phi_1 \\ \mathbf{AF}\phi_1 &= \mathbf{A}true\mathbf{U}\phi_1 \\ \mathbf{EG}\phi_1 &= \neg\mathbf{AF}\neg\phi_1 \\ \mathbf{AG}\phi_1 &= \neg\mathbf{EF}\neg\phi_1 \end{aligned}$$

Points Fixes :
$$\begin{aligned} M, s \models \mathbf{A}\phi_1\mathbf{U}\phi_2 &\Leftrightarrow M, s \models \phi_2 \vee (\phi_1 \wedge \mathbf{AX}(\mathbf{A}\phi_1\mathbf{U}\phi_2)) \\ &\Leftrightarrow s \in sat_{\phi_2} \cup (sat_{\phi_1} \cap sat_{\mathbf{AX}(\mathbf{A}\phi_1\mathbf{U}\phi 2)}) \\ &\Leftrightarrow s \in \mu Q[sat_{\phi_2} \cup (sat_{\phi_1} \cap sat_{\mathbf{AX}Q})] \\ M, s \models \mathbf{E}\phi_1\mathbf{U}\phi_2 &\Leftrightarrow M, s \models \phi_2 \vee (\phi_1 \wedge \mathbf{EX}(\mathbf{E}\phi_1\mathbf{U}\phi_2)) \\ &= s \in sat_{\phi_2} \cup (sat_(\phi_1) \cap sat_{\mathbf{EX}(\mathbf{E}\phi_1\mathbf{U}\phi_2)}) \\ &= s \in \mu Q[sat_{\phi_2} \cup (sat_{\phi_1} \cap sat_{\mathbf{EX}Q)})] \end{aligned}$$

FIG. 2.1 Opérateurs nécessaires à la vérification symbolique de propriétés CTL [EC80]

2.2 Les techniques de représentation symbolique

L'efficacité des techniques de model-checking symbolique dépend du type de données abstrait que l'on choisit pour la représentation d'ensembles d'états. Le succès rencontré par les Diagrammes de Décision Binaires (BDD, [Bry86]) a ouvert la voie à une importante zoologie de diagrammes de décision pour la représentation de fonctions booléennes. L'activité de recherche autour de ces structures a aussi donné naissance à des diagrammes voués à la représentation d'espaces d'états. Nous détaillons dans cette section les principales représentations proposées.

2.2.1 Les BDD

Les BDD [Bry86] ont connu un indéniable succès pour la vérification de systèmes finis. Ils étaient à l'origine consacrés à la représentation de fonctions booléennes. La représentation symbolique d'espace d'états basée sur la représentation de fonctions booléennes [CBM90a] s'est imposée suite à la parution de l'article *Symbolic Model Checking : 10^{20} states and beyond* [BCM+90] qui fait explicitement mention des BDD comme type de données abstrait pour la représentation d'états pour un système fini :

- Un ensemble fini d'états $S' \subseteq S$ peut être représenté par sa fonction caractéristique $\chi_{S'} : S \to \mathbb{B}, s \in S' \iff \chi_{S'}(s)$.
- Les BDD sont utilisés pour représenter des fonctions de $\mathbb{B}^n \to \mathbb{B}$. En utilisant un encodage de S dans $\mathbb{B}^n$, on peut utiliser les BDD pour représenter des ensembles d'états.

On dispose alors d'une représentation d'ensembles d'états bénéficiant des propriétés des BDD détaillées plus bas.

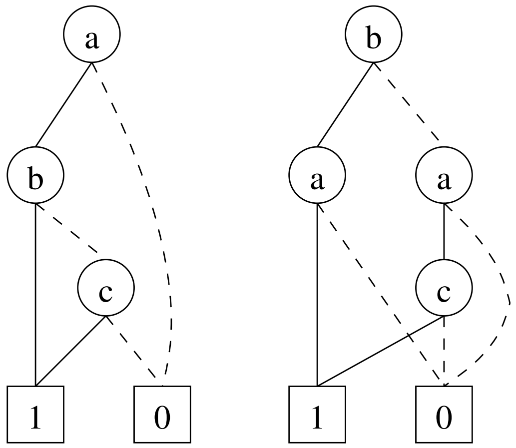


FIG. 2.2 Deux BDD ordonnés

Les BDD sont des graphes orientés acycliques (DAG) dont chaque nœud non terminal est étiqueté par une variable booléenne. La figure 2.2 montre deux BDD représentant la fonction $\phi = a \wedge (b \vee c)$ sous deux ordres différents $(a < b < c)$ et $(b < a < c)$. Chaque nœud a deux arcs sortants qui correspondent aux cas où la variable est vraie (lignes pleines) ou fausse (pointillés). Les feuilles terminales (en carrés) sont étiquetées par 1 ou 0. Elles correspondent aux deux valeurs que peut prendre la fonction. L'interprétation associant un nœud à une fonction booléenne est fondée sur la décomposition de Shannon qui décompose en x la fonction ϕ suivant les deux valeurs de la variable x (x et $\bar{x}$) et les cofacteurs associés (ϕ_x et $\phi_{\bar{x}}$) :

$$\phi \quad = \quad (\bar{x} \wedge \phi_{\bar{x}}) \vee (x \wedge \phi_x)$$

Si l'on suppose que les variables sont ordonnées, alors chaque chemin menant de la racine aux feuilles rencontre chaque variable suivant le même ordre ascendant. Ceci permet, entre autres, de ne pas représenter explicitement les nœuds dont les deux arcs sortants pointent sur le même nœud.

En appliquant des règles de réduction [Bry86] il est possible de représenter un BDD ordonné sous une forme canonique. Ainsi, les BDD ordonnés et réduits (ROBDD) représentent des fonctions booléennes sous une forme à la fois compacte (grâce aux propriétés de partage sur les nœuds) et canonique (par la suite on note BDD un BDD ordonné et réduit).

Les BDD pour représenter des opérations ensemblistes et des affectations

Les opérations logiques et les affectations de valeurs constantes peuvent être réalisées sur des ROBDD en un temps qui est une fonction polynômiale du nombre d'opérandes qui interviennent dans l'opération : Soient deux BDD bdd_1 et bdd_2 qui représentent respectivement deux fonctions ϕ_1 et ϕ_2, la fonction $apply(\diamond, bdd_1, bdd_2)$ [Bry86] construit le BDD représentant $\phi_1 \diamond \phi_2$ où $\diamond$ est un opérateur booléen binaire quelconque. La fonction $apply$ réalise un parcours attelé récursif, particularisé par l'opérateur $\diamond$, des deux BDD. Aussi, les opérations ensemblistes élémentaires (union, intersection et différence) peuvent être réalisées à l'aide des opérateurs booléens classiques ($\vee$, $\wedge$ et $\neg$) :

$$\begin{aligned}
s \in S_1 \cup S_2 &\iff \chi_{S_1}(s) \vee \chi_{S_2}(s) \\
s \in S_1 \cap S_2 &\iff \chi_{S_1}(s) \wedge \chi_{S_2}(s) \\
s \notin S_1 &\iff \neg\chi_{S_1}
\end{aligned}$$

La figure 2.3 montre comment réaliser l'affectation $a \leftarrow b$ sur l'ensemble des états de fonction caractéristique ϕ. Cette affectation se fait en trois temps :

- On abstrait d'abord la valeur de a dans la fonction ϕ : $\phi' = \phi_{|a}$. On obtient alors un sur-ensemble des états de fonction caractéristique ϕ pour lesquels a est quelconque.
- On construit l'opérateur réalisant l'égalité $a = b$ à l'aide de l'équivalence $xnor(a, b) = a\bar{\oplus}b = (a \wedge b) \vee (\neg a \wedge \neg b)$.
- On construit la conjonction des deux derniers BDD obtenus à l'aide de la fonction « apply » :

$$\begin{aligned}
\phi_{|a\leftarrow b} &= ((a \wedge b) \vee (\neg a \wedge \neg b)) \wedge \phi_{|a} \\
&= ((a \wedge b) \vee (\bar{a} \wedge \bar{b})) \wedge (\phi_a \vee \phi_{\bar{a}}) \\
&= (b \wedge (a \wedge (\phi_a \vee \phi_{\bar{a}}))) \vee (\bar{b} \wedge (\bar{a} \wedge (\phi_a \vee \phi_{\bar{a}}))) \\
&= (b \wedge \phi_{|a\leftarrow 1}) \vee (\bar{b} \wedge \phi_{|a\leftarrow 0})
\end{aligned}$$

Notons que dans le cadre de la vérification symbolique, les états *et* les transitions sont représentables sur des BDD.

L'ordonnancement des variables d'un BDD devient crucial lorsque la profondeur de celui-ci devient importante, c'est-à-dire quand la fonction booléenne qu'il représente contient beaucoup d'opérandes distincts. L'ordonnancement *idéal* d'un BDD peut changer suivant les phases de calcul de l'application qui le manipule. Le réordonnancement *dynamique* des variables [R. 93] permet en partie de repousser ces limitations.

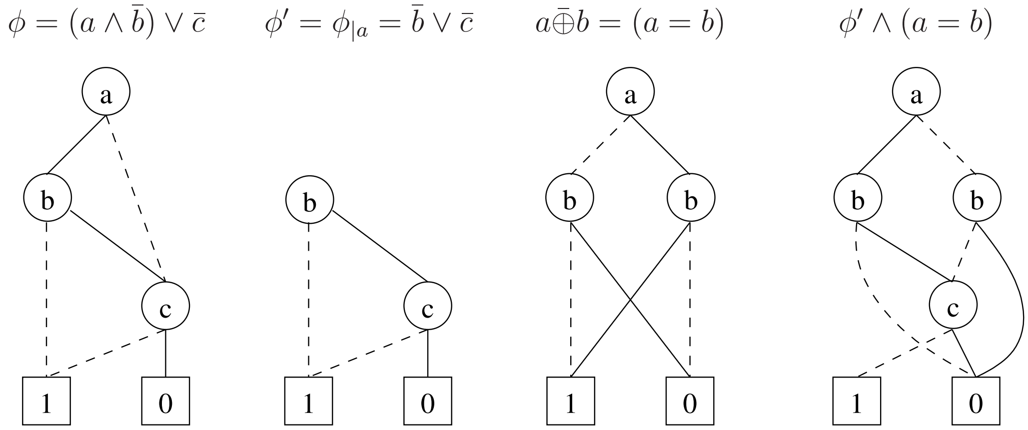


FIG. 2.3 Affectation $a \leftarrow b$ sur l'ensemble des états de fonction caractéristique $(a \wedge \bar{b}) \vee \bar{c}$

2.2.2 Structures dérivées des BDD

Le succès des BDD a ouvert de nombreuses voies de recherches portant sur la représentation de la fonction booléenne. Les principales ramifications de la famille des Diagrammes de Décision pour la vérification de systèmes sont données sur la figure 2.4 qui comporte trois sous-ensembles :

1. La branche ZBDD qui s'appuie sur les BDD pour représenter des ensembles de vecteurs de bits contenant un grand nombre de zéros [Min93].
2. Le sous-ensemble central issu des ROBDD résume l'ensemble des représentations en DAG pour des fonctions sur variables booléennes [Bry86, KSR92, DST$^+$94, CMZ$^+$93, BC95, CFZ95, LS92, TP97, BC95].
3. Le sous-ensemble issu des MDD donne les représentations en DAG pour des fonctions sur des variables qui ne sont pas nécessairement booléennes [CLS99, CS02, CEPA$^+$02, TM04].

Le tableau 2.1 synthétise les principales caractéristiques et distinctions de ces structures :
- les dix premières lignes correspondent aux deux premiers cas,
- les quatre dernières lignes correspondent aux troisième et dernier cas.

La colonne « Décomposition » décrit l'interprétation des arcs sortants des nœuds intermédiaires :
- la décompostion de Shannon qui est utilisée pour les BDD (voir section 2.2.1),
- la décomposition de Reed-Muller, basée sur l'opérateur ou-exclusif et la relation

$$\begin{aligned} f &= (f_{\bar{x}}) \oplus (x \wedge (f_{\bar{x}} \oplus f_x)) \\ &= (f_{\bar{x}}) \oplus (x \wedge f_{\delta x}) \\ &= (f_x) \oplus (\bar{x} \wedge f_{\delta x}) \end{aligned}$$

(e symbole f_δ est appelé « différence négative de la fonction f par rapport à la variable x »),
- la décomposition de Shannon étendue à des variables définies sur des domaines quelconques.

La colonne « Réduction » donne, pour un nœud intermédiaire donné, le critère indiquant s'il n'est pas nécessaire de représenter explicitement le nœud ; c'est le cas, dans un BDD par exemple, lorsque les deux cofacteurs ϕ_x et $\phi_{\bar{x}}$ sont identiques. Les formes hybrides (OKFDD et HDD) implémentant indifféremment les décompositions de Shannon ou de Reed-Muller utilisent une décomposition adaptée à la décomposition de chaque nœud ($\phi_x = \phi_{\bar{x}}$ pour Shannon et $\phi_x = 0$ pour Reed-Muller). La colonne « Pondération » donne la pondération éventuelle des arcs pour les fonctions entières.

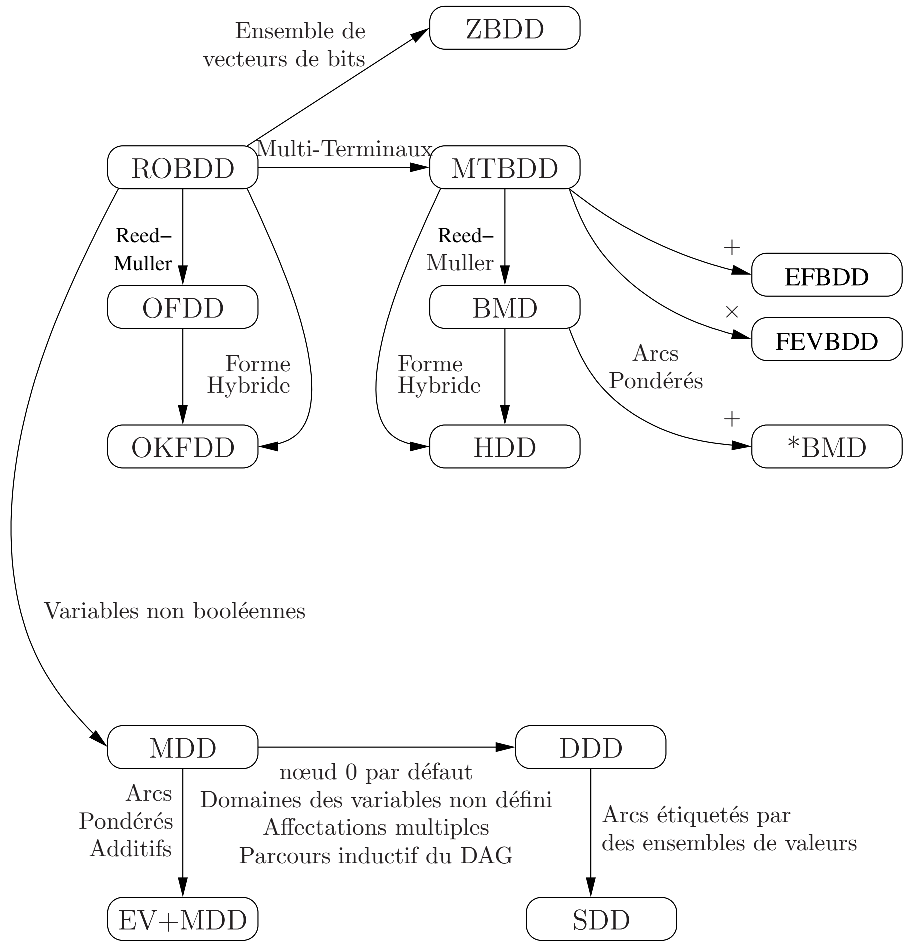


FIG. 2.4 Principales représentations de fonctions ou d'ensembles finis utilisant les DAG

Dans l'ensemble, aucune représentation ne surclasse réellement les autres : c'est la classe de fonctions à représenter qui détermine la structure la plus adaptée.

Les fonctions sur des variables entières ou quelconques sont données sur la partie basse de la figure 2.4. Elles correspondent aux quatre dernières lignes du tableau 2.1 et s'appliquent à des représentations de plus haut niveau que les précédentes car *on s'affranchit de la représentation booléenne des variables*. Les Diagrammes de Décision Multivalués (MDD, [MC99]) on été développés principalement pour la génération et le stockage d'espace d'états. Ils nécessitent néanmoins de connaître *a priori* le domaine de définition des variables de la fonction représentée.

Les fonctions encodées par un MDD sont de la forme [CLS99] :

$$f : \{0, \dots, N_K\} \times \dots \times \{0, \dots, N_1\} \rightarrow \{0, \dots M\}$$

A chaque variable $v_i, 1 \leq i \leq K$, est associée une strate où chaque nœud comporte $N_i + 1$ arcs sortants associés aux valeurs du domaine de v_i, dirigés vers les nœuds de la strate associée à la variable v_{i-1}. Le nombre de nœuds sur un chemin sera moins important que sur un BDD, la contrepartie étant un accroissement du nombre d'arcs sortants possibles pour un nœud donné.

Les MDD ont été pourvus d'un formalisme permettant de réaliser des modifications locales n'affectant que quelques strates contigües [CLS99]. Cette méthode a donné d'excellents résultats expéri-

Nom	Domaine	Image	Décomposition	Réduction	Pondération
ZDD	$\mathbb{B}^n$	$\mathbb{B}$	Shannon	$\phi_{\bar{x}} = 1$	Aucune
BDD	$\mathbb{B}^n$	$\mathbb{B}$	Shannon	$\phi_x = \phi_{\bar{x}}$	Aucune
OFDD	$\mathbb{B}^n$	$\mathbb{B}$	Reed-Muller	$\phi_x = 0$	Aucune
OKFDD	$\mathbb{B}^n$	$\mathbb{B}$	Shannon ou Reed-Muller	Selon dec.	Aucune
MTBDD	$\mathbb{B}^n$	$\mathbb{N}$	Shannon	$\phi_x = \phi_{\bar{x}}$	Aucune
EFBDD	$\mathbb{B}^n$	$\mathbb{N}$	Shannon	$\phi_x = \phi_{\bar{x}}$	Additive
FEVBDD	$\mathbb{B}^n$	$\mathbb{N}$	Shannon	$\phi_x = \phi_{\bar{x}}$	Multiplicative
BMD	$\mathbb{B}^n$	$\mathbb{N}$	Reed-Muller	$\phi_x = 0$	Aucune
*BMD	$\mathbb{B}^n$	$\mathbb{N}$	Reed-Muller	$\phi_x = 0$	Multiplicative
HDD	$\mathbb{B}^n$	$\mathbb{N}$	Shannon ou Reed-Muller	Selon dec.	Aucune
MDD	$\mathbb{N}^n$	$\mathbb{N}$	Shannon étendue	$\forall i, \phi_{x=i} = 0$	Aucune
EV+MDD	$\mathbb{N}^n$	$\mathbb{N}$	Shannon étendue	$\forall i, \phi_{x=i} = 0$	Additive
DDD	Données	$\{1, \top\}$	Shannon étendue	$\forall d, \phi_{x=d} = 0$	Aucune
SDD	Ensembles	$\{1, \top\}$	Shannon étendue	$\forall s, \phi_{x\in s} = 0$	Aucune

TAB. 2.1 Récapitulatif des représentations de fonctions sur des DAG

mentaux pour la vérification de réseaux de Petri. De plus, les MDD ont, entre autres, donné naissance aux “Additive Edge-Valued MDD” (EV+MDD, [CS02]) pour représenter de façon compacte les fonctions arithmétiques sur des ensembles de variables numériques.

Une représentation voisine des MDD, les Diagrammes de Décision de Données [CEPA$^+$02], diffère de ces derniers sur trois points :

– Les bornes des variables n’ont pas besoin d’être connues *a priori*.
– Il est possible d’ajouter *dynamiquement* de nouvelles variables.
– Ils sont pourvus d’un formalisme de définition d’opérateurs inductifs.

Les DDD sont décrits dans la section 2.3. Enfin, les Diagrammes de Décision d’Ensembles (SDD, [TM04]) reprennent le formalisme des DDD avec un étiquetage symbolique des arcs. Ils ont été définis au cours de cette thèse, nous motiverons leur emploi au chapitre 6.

2.2.3 Les Diagrammes de décision Multivalués

Les MDD ont été définis pour représenter des ensembles d’états. S’il s’avère que les MDD ne sont pas satisfaisants pour la vérification symbolique de programmes Promela, les méthodes qui leurs sont associées ont largement contribué au développement de cette thèse. Nous consacrons ici une section entière aux MDD afin de souligner trois points essentiels :

1. Le concept de représentation « orientée états », que les MDD partagent avec les DDD, et qui constituent une rupture importante avec les modes de représentation abordés jusqu’ici.
2. Le principe de « localité », bien adapté dans le cadre de l’analyse des réseaux de Petri, qui trouve ses limites pour les systèmes écrits en langage Promela.
3. Les méthodes de « saturation » qui procurent un gain en ressources matérielles significatif pour la construction de l’ensemble des états accessibles.

Les travaux de recherche sur les MDD que nous présentons dans cette section sont concomitants aux nôtres.

Structure d'un MDD

Nous considérerons par la suite que les MDD définissent des fonctions d'un sous-ensemble fini de $\mathbb{N}^n$ dans $\mathbb{B}$. Ils sont représentés par des diagrammes de décision, où chaque variable apparait une fois sur chaque chemin menant de la racine à la feuille 1 (les chemins menant à la feuille 0 ne sont pas représentés), en respectant un ordre prédéfini.

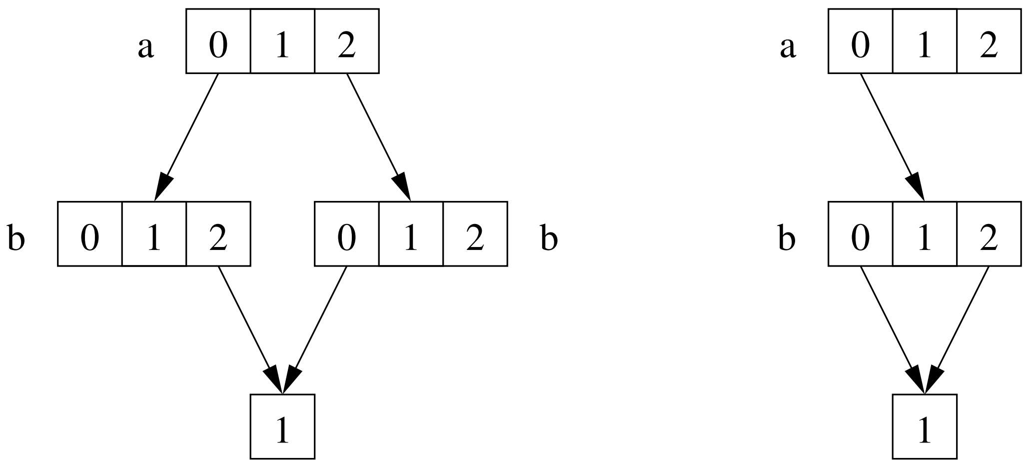


FIG. 2.5 Deux MDD

La partie gauche de la figure 2.5 donne le MDD associé à la fonction

$$f(a,b) \quad = \quad \begin{cases} 1 & \text{si}((a=0) \wedge (b=2)) \vee ((a=2) \wedge (b=0)) \\ 0 & \text{sinon} \end{cases}$$

Interprétation des MDD

Lorsqu'elles sont utilisées pour la représentation d'ensembles d'états, les approches classiques cités par [Bry95] font intervenir la notion de fonction caractéristique dont on souhaite connaître le *support* (*i.e.* l'ensemble des états pour lesquels cette fonction est vraie). Comme les chemins menant au nœud terminal 0 ne sont pas représentés sur un MDD, la représentation d'une telle fonction est un DAG dont tous les chemins mènent au nœud terminal 1. Ceci mène à l'association

$$\text{Un Chemin du DAG} \iff \text{Un Etat du Système}$$

car seul le support de la fonction caractéristique est réellement représenté.

Exemple 2.1 (interprétation de deux MDD) *La figure 2.5 montre deux MDD utilisés pour la représentation d'espaces d'états. Ils comportent deux strates (a et b), les nœuds associés à ces variables sont assimilés à des « tableaux » d'arcs implicitement étiquetés par les valeurs que peuvent prendre les variables (ici 0, 1 ou 2). Sur le MDD de gauche, on dénombre deux chemins (donc deux états), celui de gauche pour lequel $a = 0$ et $b = 2$ et celui de droite pour lequel $a = 2$ et $b = 0$. Dans ce qui* *suit, on considère que les arcs sont implicitement étiquetés par leur place dans le tableau, et que l'on dispose d'un accès « direct » aux nœuds d'une strate donnée. Par exemple, dans le cas de la figure 2.5, il ne sera pas nécessaire de suivre les arcs issus du nœud étiqueté par* a *pour accéder aux deux nœuds étiquetés par* b *(il existe un mécanisme menant directement aux nœuds de la strate* b*).*

Localité

Le fait de s'abstraire d'une construction bit à bit permet de se doter d'une représentation plus concise (plus lisible aussi) d'un ensemble d'états. Pour un nœud donné, on dispose immédiatement des valeurs prises par la variable qui lui est associée dans un contexte particulier. A partir du moment ou l'on ne raisonne plus en termes de fonctions booléennes, affecter une valeur à cette variable peut se résumer à ne conserver qu'un arc sortant étiqueté par cette valeur et dirigé vers l'union des nœuds fils. Cela suppose de disposer d'un accès rapide à la strate étiquetant la variable et de modifier les nœuds *sans modifier leur référence*.

Cette approche constitue une rupture importante avec les BDD. En effet, si l'on considère l'affectation $a \leftarrow 0$ sur un BDD et sur un MDD, les traitements à réaliser sont les suivants :

- Sur le BDD on réalise la fonction $\phi' = \phi_{|a} \wedge \neg a$, dont l'évaluation de l'opérateur $\wedge$ suppose un parcours complet du DAG.
- Cette même affectation sur MDD est donnée sur la figure 2.5 : Le MDD de gauche représente deux états $(a = 0) \wedge (b = 2)$ et $(a = 2) \wedge (b = 0)$. L'affectation $a \leftarrow 0$ mène à deux nouveaux états $(a = 0) \wedge (b = 2)$ et $(a = 0) \wedge (b = 0)$ qui sont représentés sur le MDD à droite de la figure 2.5. Ainsi, partant de l'ensemble d'états $((a = 0) \wedge (b = 2)) \vee ((a = 2) \wedge (b = 0))$, nous avons construit l'ensemble $((a = 0) \wedge (b = 2)) \vee ((a = 0) \wedge (b = 0))$ qui s'écrit $(a = 0) \wedge ((b = 0) \vee (b = 2))$ et est représenté à droite de la figure 2.5. La construction du MDD résultat peut être réalisée à partir du MDD de gauche en *remplaçant* le nœud a par un nœud n'ayant qu'un seul arc sortant, étiqueté par la valeur 0 et dirigé vers l'union des deux nœuds « fils ». Ce remplacement est réalisé sans modifier la référence du nœud a et ses arcs entrants demeurent inchangés.

Application à la construction de l'ensemble des états accessibles

Dans le cas général, ce principe utilisant la localité (*event locality* [CLS99]), s'applique aux transitions faisant intervenir un petit nombre de variables représentées sur des strates voisines. Une transition donnée est particularisée suivant toutes les valeurs que peuvent prendre les variables qui sont concernées. Chacune de ces sous-transitions est réalisée « en place » à l'aide de jeux de pointeurs comme le montre l'exemple suivant :

Exemple 2.2 (Utilisation du principe de localité) *La figure 2.6 montre comment obtenir l'ensemble des états accessibles suite à l'exécution de l'instruction* $b \leftarrow b + c$*. Au centre de la partie haute de la figure se trouve l'ensemble des états avant l'activation de la transition, les variables sont (dans l'ordre)* a*,* b*,* c*, et* d*. Le domaine de chaque variable est* $\{0, 1, 2\}$*, et chaque nœud est assimilé à un tableau de trois arcs implicitement étiquetés par l'une de ces trois valeurs (ici, suivant l'ordre croissant) et menant à la strate associée à la variable suivante.*

On souhaite unir l'ensemble des états courants à l'ensemble de leurs successeurs immédiats par la transition $t : b \leftarrow b + c$*. Cette transition ne concerne que les valeurs de* b *et* c*, les valeurs de* a *et* d

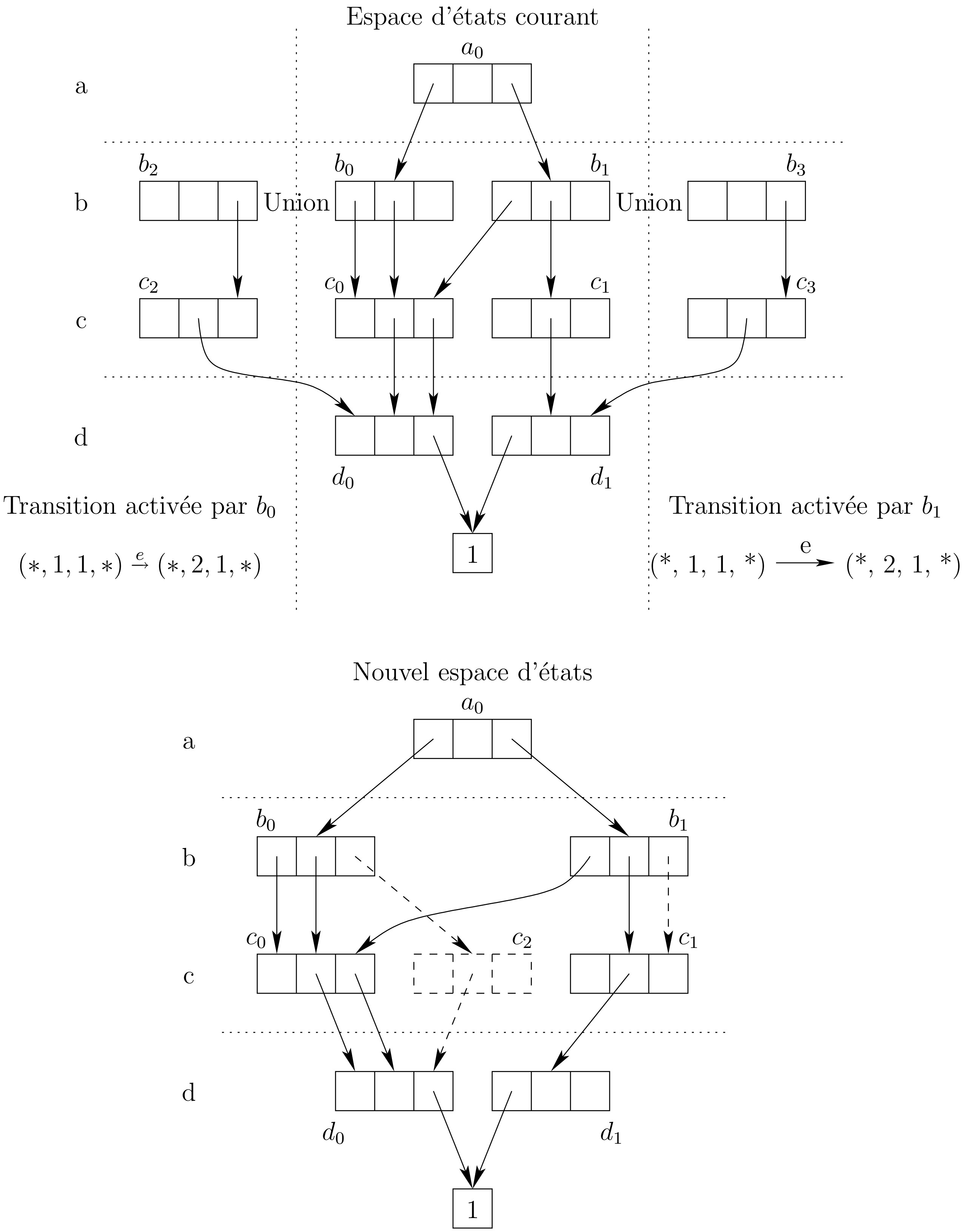


FIG. 2.6 Application *locale* d'une transition sur un MDD

n'interviennent pas : on note t_{ij} l'ensemble des transitions particularisées aux valeurs (i et j) de b et c : $t_{ij} : (*, i, j, *) \rightarrow (*, i + j, i, *)$.

Ensuite pour toutes les valeurs i et de j dans $[0, 2]$,

1. *on sélectionne les états $S_{ij} = S|_{(b=i)\wedge(c=j)}$,*
2. *on procède à l'union de $S_{ij} \cup t_{ij}(S_{ij})$:*
   - *le préfixe de la strate la plus haute (a) n'est pas modifié,*
   - *le suffixe de la strate la plus basse (d) non plus,*
   - *les nouveaux nœuds intermédiaires (strates b et c) sont construits en respectant les mécanismes classiques assurant l'unicité de représentation (par l'intermédiaire d'une table d'unicité des nœuds).*

Dans la figure 2.6, ce procédé est appliqué pour la transition $t_{11} = (*, 1, 1, *) \leftarrow (*, 2, 1, *)$.

Comme a n'est pas concernée par la transition, on conserve les préfixes $a_0 \xrightarrow{0}$ et $a_0 \xrightarrow{2}$ et le calcul commence à la strate b. Les chemins activant la transition sont : $b_0 \xrightarrow{1} c_0 \xrightarrow{1} [d_0]$ et $b_1 \xrightarrow{1} c_1 \xrightarrow{1} [d_1]$ (on a noté entre crochets les suffixes qui ne sont pas modifiés). On unit ces chemins avec les motifs des ensembles résultats auxquels on a ajouté les suffixes des états activant la transition, à savoir $b_2 \xrightarrow{2} c_2 \xrightarrow{1} [d_0]$ et $b_3 \xrightarrow{2} c_3 \xrightarrow{1} [d_1]$ situés aux « ailes » de la figure. Le nœud c_3 est identique à c_1, et possède donc la même référence : l'union de b_1 avec b_3 conduit à l'ajout d'un nouvel arc sortant issu de b_1. Le nœud c_2 est différent de c_0 : un nouveau chemin, issu de b_0 est ajouté. Toutes ces modifications sont réalisées en place à l'aide de jeux de pointeurs.

Cette méthode a largement fait ses preuves pour la génération d'espace d'états accessibles sur des réseaux de Petri où l'on dispose d'une forte localité pour les transitions, de domaines de variables petits et contrôlés, et d'un ordonnancement des variables *ad hoc*.

Dans le cadre de systèmes décrits en Promela, l'emploi de MDD pose quatre problèmes majeurs :

1. On ne connait qu'un sur-ensemble du domaine de chaque variable par l'intermédiaire du type C utilisé pour leurs déclarations. Le risque avec ce type de représentation est de construire des tableaux de trop grande taille (et donc occupant une place démesurée) pour une variable mal dimensionnée.
2. L'emploi de variables partagées et de canaux en Promela tend à invalider l'hypothèse de localité.
3. La particularisation énumérative des expressions pâtit des deux points précédents, au point de devenir un facteur supplémentaire d'explosion combinatoire dans le cas d'expressions contenant beaucoup de variables aux domaines trop grands.
4. Les modifications « en place » ne permettent pas de manipuler plusieurs ensembles d'états distincts, toute modification sur un nœud se répercutant sur tous les DAG qui le contiennent.

Nous verrons en quoi les DDD, et les méthodes qui leur sont associées, abolissent ces problèmes par une représentation plus concise des opérateurs.

Saturation des strates

Le principe de saturation [CLS01, CC04] n'est réellement efficace qu'avec des méthodes exploitant la localité des transitions, le mécanisme de saturation permet d'exploiter plus encore la localité des transitions.

Jusqu'ici, la génération de l'espace des états accessibles se faisait en appliquant toutes les transitions possibles à un ensemble d'états pour n'obtenir que les successeurs immédiats. L'algorithme de principe est le suivant :

Algorithme 2.1 : Génération de l'ensemble des états accessibles par une recherche en largeur d'abord $ReachBFS(S_0, T)$

Input : Un ensemble d'états initiaux S_0
Input : Un ensemble de transitions T
Data : Des ensembles intermédiaires d'états S_1 et S_2
Output : L'ensemble des états accessibles

$S_1 \leftarrow \emptyset$
$S_2 \leftarrow S_0$
while $S_1 \neq S_2$ **do**
  $S_1 \leftarrow S_2$
  foreach $t \in T$ **do**
    $S_2 \leftarrow S_2 \cup post_t(S_1)$
return S_2

Le principe de saturation se base sur les observations suivantes :

- l'ordre de franchissement des transitions n'est pas significatif (on obtiendra le même ensemble d'états accessibles)
- il existe des ordres de franchissement des transitions (qui regroupent celles qui interviennent sur les mêmes strates) permettant de tirer parti de la localité et d'améliorer le calcul du point fixe.

L'algorithme de principe est le suivant :

Algorithme 2.2 : Génération de l'ensemble des états acessibles par saturation $ReachSat(S_0, T)$

Input : Un ensemble d'états initiaux S_0
Input : L'ensemble V des variables caractérisant le système
Input : Un ensemble de transitions T. On note T_v l'ensemble des transitions sensibles à la variable v
Data : Des ensembles intermédiaires d'états S_1, S_2
Output : L'ensemble des états accessibles

$S_1 \leftarrow \emptyset$
$S_2 \leftarrow S_0$
while $S_1 \neq S_2$ **do**
  $S_1 \leftarrow S_2$
  foreach $v \in V$ **do**
    $S_2 \leftarrow S_2 \cup ReachBFS(S_1, T_v)$
return S_2

Ce principe repose grandement sur la localité des transitions, après les avoir classées suivant la profondeur de leur application sur l'arbre (par exemple, des strates les plus basses vers les strates les plus hautes), on les applique jusqu'à l'obtention d'un point fixe (on dit alors que les strates associées sont saturées), avant d'appliquer les transitions concernant d'autres strates.

Un exemple illustrant l'avantage des méthodes de saturation est donné dans l'annexe A.

2K-MDD

Les principaux avantages des MDD sont l'accès rapide aux strates à modifier et la représentation extrêmement concise de leurs nœuds qui nécessite, en machine, un espace mémoire restreint. Les inconvénients sont la nécessité de connaître *a priori* le domaine des variables, de particulariser les transitions et les modifications « en place » imposent de ne travailler que sur un unique ensemble d'états. Les travaux récents de Gianfranco Ciardo *et al.* [CY05] abolissent ces derniers griefs par une représentation des transitions sur un système de k variables à l'aide d'un MDD de $2k$ variables (2KMDD), alternant valeurs d'activation et valeurs résultantes ainsi que le critère local de réduction. L'accès aux strates n'est plus imédiat mais les transitions n'étant plus particularisées, il s'ensuit un gain en performances très appréciable. Mieux, cette méthode s'applique aussi sur des domaines de variables non bornés et permet de manipuler *plusieurs* ensembles d'états.

2.3 Les Diagrammes de Décision de Données

Les Diagrammes de Décision de Données sont des DAG très voisins des MDD. Ils respectent le même principe de représentation « orientée états » de ces derniers mais le domaine des variables ainsi que le nombre de nœuds sur un chemin n'est pas figé. De plus la réalisation des opérateurs sur DDD se fait à l'aide de mécanismes de parcours récursifs particularisés (dits inductifs) qui diffèrent des modifications « en place » que proposent les MDD. Nous définissons dans cette section les DDD et leurs opérateurs, présentés dans [CEPA$^+$02]. Toutes ces définitions sont indispensables pour la compréhension des contributions présentées aux chapitres 4 et 6.

2.3.1 Définition

Les diagrammes de décision de données (DDD) sont des structures en DAG qui représentent des séquences d'affectations. Les nœuds sont étiquetés par les variables et les arcs par les valeurs affectées. Aucun ordonnancement de variable n'est supposé et une même variable peut apparaître plusieurs fois sur un même chemin. Les séquences acceptées ont le nœud 1 comme feuille terminale. Le nœud 0 est la valeur par défaut et correspond aux séquences qui ne sont pas acceptées (ces séquences ne sont pas représentées). La feuille $\top$ caractérise les séquence indéfinies. La figure 2.7 montre deux

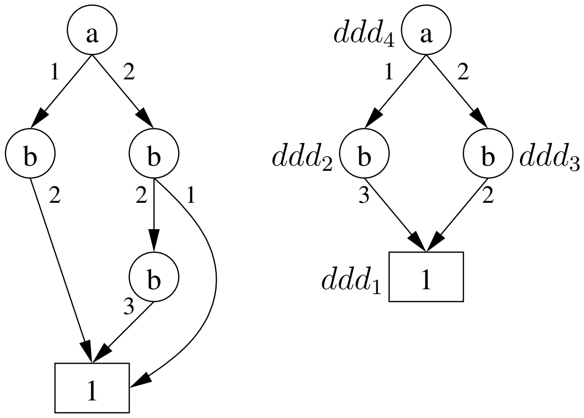


FIG. 2.7 Deux DDDs

DDD qui correspondent aux ensembles d'affectations suivantes :

1. Pour le DDD de gauche :
   - $\{a = 1; b = 2; \}\cup$
   - $\{a = 2; b = 1; \}\cup$
   - $\{a = 2; b = 2; b = 3; \}$
2. Pour le DDD de droite :
   - $\{a = 1; b = 3; \}\cup$
   - $\{a = 2; b = 2; \}$

Par la suite nous appellerons E un ensemble de variables, et pour chaque variable e, $Dom(e)$ représentera le domaine de valeurs de e.

Définition 2.3 (Diagrammes de Décision de Données DDD [CEPA+02]) *L'ensemble $\mathbb{D}$ des DDD est défini par $d \in \mathbb{D}$ si :*

- $d \in 0, 1, \top$ *ou*
- $d = (e, \alpha)$ *avec*
  - $e \in E$
  - α *est une fonction de* $Dom(e)$ *dans* $\mathbb{D}$ *telle que* $\{x \in Dom(e), \alpha(x) \neq 0\}$ *est fini.*

On note $e \xrightarrow{a} d$ *le DDD* (e, α) *tel que* $\alpha(a) = d$ *et* $\forall x \neq a$, $\alpha(x) = 0$.

Exemple 2.3 *Sur le DDD de droite de la figure 2.7, nous dénombrons quatre nœuds :*

1. *Le nœud (ou la feuille)* 1 *que l'on note* $ddd_1 = 1$
2. *Le nœud « b » de gauche* $ddd_2 = b \xrightarrow{3} 1 = b \xrightarrow{3} ddd_1$ *qui est un couple* (e_2, α_2) *pour lequel :*
   - $e_2 = b$
   - $\alpha_2(3) = ddd_1$ *et* $\forall x \neq 3, \alpha_2(x) = 0$
3. *Le nœud « b » de droite :* $ddd_3 = (b, \alpha_3)$ *où* $\alpha_3(2) = ddd_1$ *et* $\forall x \neq 2, \alpha_3(x) = 0$.
4. *Le nœud « a » :* $ddd_4 = (a, \alpha_4)$ *où*
   - $\alpha_4(1) = ddd_2$
   - $\alpha_4(2) = ddd_3$
   - $\forall x, (x \neq 1) \wedge (x \neq 2), \alpha_4(x) = 0$

Les séquences d'affectations qui ne sont pas acceptées *ne sont pas représentées* sur le DDD. [CEPA+02] a montré qu'ainsi, les DDD sont une structure canonique pour la représentation d'*ensembles de séquences d'affectations*. La représentation de séquences indéfinies mène à des définitions de « bonne définition » et de relations d'ordre [CEPA+02]. Nous ne retenons que la suivante : *un DDD est bien défini s'il ne contient aucune feuille terminale* $\top$. Nous verrons au chapitre 4 comment utiliser les DDD comme représentation canonique d'*ensembles d'états* et préserver leur bonne définition.

2.3.2 Opérateurs

Les DDD sont pourvus d'un formalisme d'opérations ensemblistes.

Définition 2.4 (Opérations ensemblistes sur DDD [CEPA+02]) *La somme $+$, le produit $\times$, et la différence $\setminus$ de deux DDD sont définis inductivement comme suit :*

Pour l'addition :

$+$	$d \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$d \equiv 0$	0	1	$\top$	(e_2, α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	$\top$	$\top$	$(e_1, \alpha_1 + \alpha_2)$ $si\ e_1 = e_2$ $\top$ $si\ e_1 \neq e_2$

Pour le produit :

$\times$	$d \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$d \equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	0	0	$\top$	$(e_1, \alpha_1 \times \alpha_2)$ $si\ e_1 = e_2$ 0 $si\ e_1 \neq e_2$

Pour la différence :

$\backslash$	$d \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$d \equiv 0$	0	0	0	0
1	1	0	$\top$	1
$\top$	0	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	(e_1, α_1)	$\top$	$(e_1, \alpha_1 \backslash \alpha_2)$ $si\ e_1 = e_2$ (e_1, α_1) $si\ e_1 \neq e_2$

Où pour chaque opérateur $\diamond \in \{+, \times, \backslash\}$, $\alpha_1 \diamond \alpha_2$ est l'application de $Dom(e1) \to \mathbb{D}$ avec $\forall x \in Dom(e1), (\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$.

L'algorithme de calcul de $d_1 \diamond d_2$ procède donc en trois étapes :

1. Si $d_1 \diamond d_2$ a déjà été calculé, on récupère le DDD résultat dans le cache de calcul.
2. Si d_1 ou d_2 est une feuille (0, 1 ou $\top$) ou s'il s'agit de deux nœuds intermédiaires étiquetés par des variables différentes, on renvoie le résultat donné dans le tableau de la définition 2.4. On insère ce résultat dans le cache de calcul.
3. Si $d_1 = (e, \alpha_1)$ et $d_2 = (e, \alpha_2)$ sont deux nœuds intermédiaires étiquetés par la même variable, alors *pour toutes les valeurs* x_i de $Dom(e)$, on construit le DDD $e \xrightarrow{x_i} (\alpha_1(x_i) \diamond \alpha_2(x_i))$, l'opérateur se propage donc aux strates inférieures. Le résultat est l'union de tous ces DDD intermédiaires : $\bigcup_i e \xrightarrow{x_i} (\alpha_1(x_i) \diamond \alpha_2(x_i))$. On insère ce résultat dans le cache de calcul.

On remarque que le résultat d'une opération ensembliste sur deux DDD bien définis n'est pas nécessairement bien défini.

Exemple 2.4 (Opérateur $+$) *La somme des deux DDD de la figure 2.7 est donnée sur la figure 2.8. Par souci de clarté nous avons représenté plusieurs fois le nœud 1 dans les étapes de calcul.*

La somme de d_1 et d_5 commence par la propagation de l'opérateur $+$ aux strates inférieures

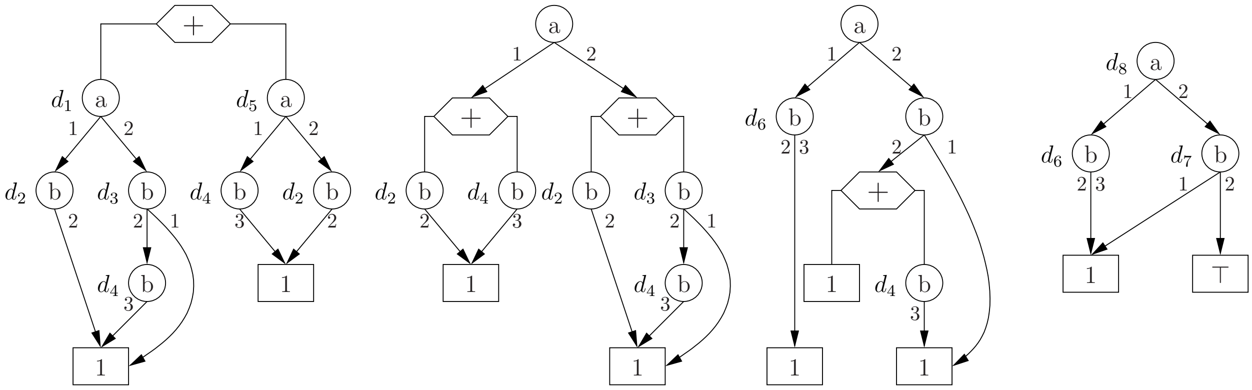


FIG. 2.8 Somme de deux DDD

(deuxième schéma) :

$$\begin{aligned} d_1 &= (a \xrightarrow{1} d_2) + (a \xrightarrow{2} d_3) \\ d_5 &= (a \xrightarrow{1} d_4) + (a \xrightarrow{2} d_2) \\ d_1 + d_5 &= (a \xrightarrow{1} (d_2 + d_4)) + (a \xrightarrow{2} (d_3 + d_2)) \end{aligned}$$

La somme $d_2 + d_4 = (b \xrightarrow{2} 1) + (b \xrightarrow{3} 1)$ donne $d_6 = b \xrightarrow{2,3} 1$ et la somme $d_3 + d_2$ nécessite une propagation supplémentaire (troisième schéma) :

$$\begin{aligned} d_2 &= b \xrightarrow{2} 1 \\ d_3 &= (b \xrightarrow{2} d_4) + (b \xrightarrow{1} 1) \\ d_2 + d_3 &= (b \xrightarrow{2} (1 + d_4)) + (b \xrightarrow{1} 1) \end{aligned}$$

La somme $1 + d_4$ est indéterminée ($\top$), la somme $d_2 + d_3$ donne $d_7 = (b \xrightarrow{2} \top) + (b \xrightarrow{1} 1)$. Le DDD résultat $d_8 = d_1 + d_5 = (a \xrightarrow{1} d_6) + (a \xrightarrow{2} d_7)$ est donné dans le dernier schéma.

En pratique, lorsqu'on utilise les DDD pour la représentation d'ensembles d'états, les opérateurs $+$, $\times$ et $\setminus$ réalisent les opérations ensemblistes élémentaires d'union, d'intersection et de différence. Nous proposons au chapitre 4 un environnement de construction des DDD pour éviter l'apparition de séquences indéfinies ($\top$).

L'opérateur de concaténation est défini en tenant compte des séquences indéfinies et correspond à la concaténation de la théorie des langages.

Définition 2.5 (Opérateur de concaténation [CEPA+02]) *Soient deux DDD d_1 et d_2, la concaténation $d_1 \cdot d_2$ est définie inductivement comme suit :*

$$d_1 \cdot d_2 = \begin{cases} 0 & \text{si } d_1 = 0 \vee d_2 \equiv 0 \\ d_2 & \text{si } d_1 = 1 \\ \top & \text{si } d_1 = \top \wedge d_2 \not\equiv 0 \\ (e, \sum_{x \in Dom(e)} (e \xrightarrow{x} (\alpha(x) \cdot d_2))) & \text{si } d_1 = (e, \alpha) \end{cases}$$

En d'autres termes, la concaténation $d_1 \cdot d_2$ « remplace » la feuille terminale 1 de d_1 par d_2. La dernière équation ($(e, \sum \ldots$) donne le mécanisme inductif pour atteindre les feuilles de d_1 : L'opérateur traverse d_1 sans modifier l'étiquetage des nœuds et des arcs rencontrés et les nœuds fils seront remplacés par le résultat de leur concaténation avec d_2.

Les opérations ensemblistes élémentaires ainsi que l'opérateur de concaténation sont ainsi définis localement et sont pourvus de mécanismes récursifs de propagation appelés « méthodes inductives ». La généralisation des méthodes inductives, proposée par [CEPA$^+$02], fait intervenir le concept « d'homomorphismes » qui seront abordés tout au long des chapitres 4 et 6.

Le formalisme associé aux homomorphismes est proposé par [CEPA$^+$02] pour modifier les séquences encodées sur un DDD. Si l'on assimile une séquence à un vecteur de données, de telles modifications peuvent être considérées comme des homomorphismes au sens algébrique. Cependant, la notion de séquence indéfinie impose une définition particulière pour les homomorphismes sur DDD :

Définition 2.6 (Homomorphismes sur DDD [CEPA$^+$02]) *Soient deux DDD d_1 et d_2 bien définis, une application Φ sur DDD est un homomorphisme si $\Phi(0) = 0$ et $\Phi(d_1 + d_2) = \Phi(d_1) + \Phi(d2)$.*

On vérifie que la somme et la composition de deux homomorphismes sont des homomorphismes :

Proposition 2.7 (Somme et composition d'homomorphismes [CEPA$^+$02]) *Soient Φ_1 et Φ_2 deux homomorphismes, alors $\Phi_1 + \Phi_2$ et $\Phi_1 \circ \Phi_2$ sont des homomorphismes.*

Le cadre donné par la définition 2.6 ne permet pas de construire de modification sur des séquences quelconques autrement que par une manipulation des opérateurs d'addition et de composition. Considérons maintenant un DDD $d = (e, \alpha)$, la définition 2.4 de l'opérateur $+$ donne : $d = (e, \alpha) = \sum_{x \in Dom(e)} e \xrightarrow{x} \alpha(x)$. La définition définit un homomorphisme par l'intermédiaire de traitements particularisés aux triplets $(e, x, \alpha(x))$ en s'appuyant sur cette propriété additive :

Proposition 2.8 (Homomorphismes inductifs [CEPA$^+$02]) *Soient c un DDD et $\phi(e, x)_{e \in E, x \in Dom(e)}$ une famille d'homomorphismes, alors*

$$\Phi(d) = \begin{cases} 0 & si\ d = 0 \\ \top & si\ d = \top \\ c & si\ d = 1 \\ \sum_{x \in Dom(e)} \phi(e, x)(\alpha(x)) & si\ d = (e, \alpha) \end{cases}$$

est un homomorphisme.

La dernière expression ($\sum_{x \in Dom(e)} \phi(e, x)(\alpha(x))$) ne s'applique qu'aux nœuds internes du DDD. On ne dispose alors que d'informations locales : La variable e concernée et ses valeurs possibles (le support de α). On applique, sur chaque DDD $\alpha(x)$, désigné par l'arc étiqueté par x sortant du nœud étiqueté par e, une fonction $\phi(e, x)$. On réunit alors les DDD obtenus. A l'instar des opérations ensemblistes, les homomorphismes inductifs peuvent faire l'objet d'une évaluation paresseuse qui peut limiter considérablement les redondances de calcul. Les homomorphismes inductifs ont la propriété remarquable d'être définis *localement*, l'homomorphisme $\phi(e, x)$ s'appliquant sur $\alpha(x)$.

Exemple 2.5 (Affectations de constantes [CEPA$^+$02]) *Prenons le cas de l'affectation constante $var = cst$, On construit l'homomorphisme $\langle$ setCst $(var, cst) \rangle$ comme suit :*

- *Les nœuds étiquetés par une autre variable que var ne sont pas concernés par l'affectation. On reporte le même homomorphisme ($\langle$ this $\rangle$) sur chacun des arcs sortants sans autre modification. Ainsi, si $e \neq var$,*

$$\begin{aligned} \langle \mathsf{setCst}\ (var, cst) \rangle (e \neq var, x) &= e \xrightarrow{x} \langle \mathsf{setCst}\ (var, cst) \rangle \\ &= e \xrightarrow{x} \langle \mathsf{this} \rangle \end{aligned}$$

– *En revanche, si* $e = var$ *tous les arcs sortants doivent être étiquetés par la valeur* cst *et les nœuds fils ne sont pas affectés ; on les reproduit donc à l'identique. On note :*

$$\langle\ \mathsf{setCst}\ (e = var, cst)\ \rangle(var, x) = e \xrightarrow{cst} \langle\ \mathsf{id}\ \rangle$$

– *Enfin, appliquer ce traitement au noeud* 1 *reviendrait à affecter une valeur à une variable non déclarée :*

$$\langle\ \mathsf{setCst}\ (var, cst)\ \rangle(1) = \top$$

Ainsi, l'affectation $c \leftarrow 1$ *sur le DDD représentant l'état* $(a = 1) \wedge (b = 2) \wedge (c = 3) \wedge (d = 4) \wedge (e = 5)$ *se réalise comme suit :*

– *Le DDD initial est* $a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1$.
– *L'homomorphisme d'affectation* $c \leftarrow 1$ *s'écrit* $\langle\ \mathsf{setCst}\ (c, 1)\ \rangle$.
– *Ce dernier se propage jusqu'à ce que la variable* c *soit atteinte :*

$$\begin{aligned} & \langle\ \mathsf{setCst}\ (c,1)\ \rangle(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\ = \quad & a \xrightarrow{1} \langle\ \mathsf{setCst}\ (c,1)\ \rangle(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\ = \quad & a \xrightarrow{1} b \xrightarrow{2} \langle\ \mathsf{setCst}\ (c,1)\ \rangle(c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \end{aligned}$$

– *L'homomorphisme remplace les valeurs de* c *par la valeur affectée (*1*), et propage l'homomorphisme d'identité :*

$$\begin{aligned} & a \xrightarrow{1} b \xrightarrow{2} \langle\ \mathsf{setCst}\ (c,1)\ \rangle(c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\ = \quad & a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{1} \langle\ \mathsf{id}\ \rangle(d \xrightarrow{4} e \xrightarrow{5} 1) \\ = \quad & a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{1} d \xrightarrow{4} e \xrightarrow{5} 1 \end{aligned}$$

L'état résultant est bien $(a = 1) \wedge (b = 2) \wedge (c = 1) \wedge (d = 4) \wedge (e = 5)$

Nous verrons comment réaliser des opérations plus complexes au chapitre 4.

2.3.3 La bibliothèque utilisée

Nous avons utilisé la bibliothèque [DDD] pour la mise en œuvre de nos prototypes. Cette bibliothèque repose sur trois modules :

1. Un module (`DDD`) de gestion des DDD qui assure le partage de la structure arborescente par une table d'unicité des nœuds du DAG.
2. Un module (`Hom`) de gestion des homomorphismes, qui utilise aussi une table d'unicité en tenant compte des données associées ainsi que des méthodes d'évaluation des homomorphismes.
3. Un module (`DED`) de calcul, qui implémente l'évaluation des opérations ensemblistes et des homomorphismes sur les DDD. Il utilise un cache d'opérations pour accélérer les calculs.

Le fonctionnement global du cache de calcul pour l'opération $d_1 \diamond d_2, \diamond \in \{+, \times, \setminus\}$ est le suivant :

– Le triplet $(\diamond, d_1, d_2)$ a déjà été inséré dans le cache de calcul, alors $d_1 \diamond d_2$ a déjà été évalué et le module `DED` renvoie le résultat de l'opération.
– Dans le cas contraire, $d_1 \diamond d_2$ est évalué de façon inductive, en utilisant le cache pour les suffixes, et le résultat est inséré dans le cache.

L'évaluation de l'application d'un homomorphisme sur un DDD fait l'objet d'un traitement similaire, à l'aide de la référence unique de l'homomorphisme évalué.

Ainsi, l'application d'un opérateur ensembliste ou d'un homomorphisme se traduit par la création, dans le cache de calcul (ensemble des DED) d'un nœud. Ce procédé qui facilite l'évaluation paresseuse des opérateurs a pour but d'accélérer les calculs effectués. La contrepartie est une occupation de la mémoire de plus en plus importante à mesure que les nœuds sont créés.

Le choix du langage d'implémentation (C++) de la bibliothèque [DDD] est lié à la gestion en mémoire de tous ces nœuds. Le langage C++ laissant à la charge du programmeur les méthodes de création et de destruction des objets, la bibliothèque a été pourvue d'un ramasse-miettes qui libère la mémoire des objets non référencés.

Précisons que les tables employées ne sont pas munies de mécanisme de réallocation ce qui fait que l'occupation en mémoire des tables associées aux trois modules est croissante même après la destruction des nœuds non référencés. Il s'en suit une occupation asymptotique de la mémoire vive en fonction du temps. La limite étant plus basse si le ramasse-miettes est souvent utilisé.

2.3.4 Forces et faiblesses des DDD

Nous étudierons en détail l'efficacité des DDD aux chapitres 4 et 6. Néanmoins nous avons isolé quelques caractéristiques qui ont orienté notre choix *a priori* et d'autres qui ont conditionné les développements à suivre.

Implémentation

Au premier abord, il résulte de l'implémentation des DDD de la bibliothèque [DDD] (à l'aide de classes C++) que l'occupation en mémoire sera bien supérieure à ce qu'occuperait, pour un même ensemble d'états, un MDD optimisé « au bit près ». De plus, la définition inductive des opérateurs peut sembler contre-performante en comparaison avec les modifications « en place » que procurent les MDD. En effet, les traitements doivent se propager depuis la racine alors que l'on pourrait accéder directement aux strates concernées et l'évaluation de tout opérateur, sur tous les nœuds d'un chemin, produit un nouveau DDD résultat. Ceci suppose des recherches répétées dans la table d'unicité pour garantir que le nouveau DDD créé référence bien un nœud unique.

Néanmoins les mécanismes mis en place dans la bibliothèque [DDD] sont transparents pour le programmeur et autorisent la manipulation simultanée de *plusieurs* DDD différents, mêmes s'ils référencent les mêmes nœuds intermédiaires.

Utilisation pour la vérification de programmes Promela

Pour la représentation d'espace d'états, les DDD sont pourvus des opérateurs ensemblistes élémentaires d'union, d'intersection et de différence dont nous avons besoin. De plus, les homomorphismes apportent un cadre élégant pour la définition d'opérateurs quelconques, sans effet de bord et tirant parti des propriétés de partage que procure la bibliothèque [DDD].

Les méthodes inductives sont, en outre, adaptées aux apparitions multiples de nœuds étiquetés par une même variable sur un même chemin. Or, cette possibilité qu'offrent les DDD est très utile pour représenter les FIFOs [CEPA$^+$02], bien qu'elle puisse provoquer l'apparition de feuilles $\top$ (voir l'exemple 2.4).

Enfin, d'une façon générale, *nous ne disposons pas encore d'une représentation canonique*, puisque les DDD $a \xrightarrow{0} b \xrightarrow{0} 1$ et $b \xrightarrow{0} a \xrightarrow{0} 1$ représentent deux séquences d'affectations différentes menant au même état $(a = 0) \wedge (b = 0)$ bien qu'ils ne soient pas équivalents et que leur union mène à une séquence indéterminée ($\top$).

En résumé, les DDD nous sont apparus comme les plus adaptés pour la représentation d'espaces d'états de système décrits en langage Promela. Pour la vérification de propriétés CTL, il faudra les munir des opérateurs $select$, $post$ et pre qui seront réalisés à l'aide d'homomorphismes inductifs. La « bonne définition » et la canonicité de ces derniers n'étant pas acquises, il faudra aussi élaborer un cadre théorique pour les garantir. Nous verrons au chapitre 4 comment résoudre ce problème.

2.4 Conclusion

Le model-checking symbolique se fonde sur un type de données abstrait pour une représentation compacte d'ensemble d'états. Les opérations sur ce type sont les opérations ensemblistes élémentaires (union, intersection et différence). Il faut ajouter la sélection d'états satisfaisant une condition booléenne ($select$). Enfin, le type doit être muni d'une opération de calcul des états successeurs ($post$) et d'une autre de calcul des états prédécesseurs (pre). Ces deux dernières doivent respecter la relation de transition du système. Nous avons choisi les Diagrammes de Décision de Données qui se distinguent des autres modes de représentation par une définition inductive de leurs opérateurs. Les propriétés de « bonne définition » et de canonicité de ces derniers n'étant pas assurées, il nous faut élaborer un cadre formel pour leur construction. Il nous faut aussi déterminer quels traitements élémentaires seront nécessaires pour réaliser les opérateurs liés à l'exécution des instructions d'un système écrit en Promela.

Chapitre 3

Sous-ensemble de Promela

Nous avons choisi d'appliquer des méthodes de vérification symbolique sur des systèmes écrits dans un sous ensemble du langage Promela suffisant pour contenir DPN. Dans ce chapitre, nous décrivons la sémantique du langage Promela et en isolons les caractéristiques communes à celles du langage DPN. Nous décrivons enfin le sous-ensemble statique que nous considérerons par la suite.

Sommaire

3.1 Le langage Promela

3.1.1 Introduction

Le langage Promela permet de décrire un réseau fini dynamique de processus asynchrones. Les communications entre processus sont réalisées par l'intermédiaire de variables partagées ou de canaux synchrones ou asynchrones. L'ensemble des canaux, des variables et des processus définit le comportement du système. SPIN [Hol97b] est un interpréteur Promela. Il permet de simuler ces systèmes en procédant par des choix aléatoires quant à l'ordre d'activation des processus, ou de branchements non exclusifs. Le processus activé exécute alors, *de façon atomique*, une instruction suivant l'état de son compteur ordinal. Les processus étant concurrents et asynchrones, plusieurs instructions peuvent être exécutables pour un état du système donné. Dans le cadre de la vérification de propriétés, toutes les branches de l'arbre d'exécution sont parcourues.

3.1.2 Les variables et leurs types

Il existe trois genres de variables en Promela : les variables scalaires, les tableaux de valeurs et les canaux. Dans tous les cas, ces variables sont typées. On retrouve les types entiers autorisés par le langage C (voir tableau 3.1.2) ainsi que les énumérations. Comme la vérification de propriétés LTL est effectuée par un programme C généré par l'interpréteur SPIN, l'espace mémoire effectif occupé dépend du compilateur C utilisé, mais en règle générale, l'unité minimale est l'octet. Ainsi, les variables encodées sur un nombre de bits différent de 8 sont soit encodées une à une sur un octet, soit regroupées sur un ou plusieurs octets. Les variables non signées dont la taille est définie, au bit près, par l'utilisateur (`unsigned`) sont utiles pour limiter les ressources nécessaires lorsque l'on connait leur domaine de définition. La contrepartie est qu'elles ne peuvent être transférées *via* un canal de communication lorsque l'on utilise SPIN pour la vérification.

nom	taille		domaine
`bit/bool`	non signé	1 bit	$0,1$
`byte`	non signé	1 octet	$0..255$
`short`	signé	2 octets	$-2^{15}..2^{15}-1$
`int`	signé	4 octets	$-2^{31}..2^{31}-1$
`unsigned`	non signé	x bits	$0..2^{x}-1$

TAB. 3.1 Types de bases proposés en Promela

En outre Promela accepte les types structurés définis par l'utilisateur, ceux-ci peuvent à leur tour instancier des champs dont le type est prédéfini ou déclaré par l'utilisateur. Les tableaux et les canaux sont de taille fixe, et les références sont interdites.

Exemple 3.1 (Types et Variables Promela) *Considérons le code Promela suivant :*

```
typedef Header {
  int source;
  int dest;
  int id;
}
typedef Message{
  Header head;
  int data[4];
}

typedef Acknowledgement {
  bool status;
```

```
  int id;
}

Message Table[5];
chan StoR = [3] of Message;
chan RtoS = [0] of Acknowledgement;
```

Le code ci-dessus définit trois types de données :

- *Les en-têtes (`Header`) contenant l'identifiant de la source (`source`), de la destination (`dest`) et du message (`id`).*
- *Les messages (`Message`) contenant des données entières (`data`) rangées dans un tableau de 4 entiers et un en-tête (`head`).*
- *Les acquittements (`Acknowledgement`) contenant un statut booléen (`status`) et l'identifiant du message acquitté (`id`).*

Les trois dernières lignes montrent l'utilisation de ces types sous la forme d'un tableau de 5 messages (`Table`) et de deux canaux de communication. Le premier canal (`StoR`) peut contenir jusqu'à trois messages, on parle alors de canal asynchrone (la taille du buffer est spécifiée entre crochets). Le second (`RtoS`) permet de transférer des acquittements sous forme de communication par rendez-vous (un processus ne pourra écrire dans ce canal que si, au même instant, un autre processus tente de lire la donnée, on parle alors de canal synchrone).

La portée d'une variable peut être limitée à un seul processus si elle est déclarée au sein de celui-ci (voir la section 3.1.3), dans le cas contraire sa portée est étendue à tous les processus.

Les canaux sont généralement déclarés en tant que variables globales. Néanmoins la syntaxe Promela autorise la déclaration d'un canal au sein d'un processus qui pourra en transmettre la référence aux processus qu'il instancie par l'intermédiaire de l'instruction `run` (voir section 3.2.1). Ce procédé introduit une forme de transmission de variables par référence dont la portée est réduite à une famille de processus, le père étant le processus déclarant la variable.

Quel que soit le mode de déclaration d'un canal, il est possible de donner à un processus l'exclusivité en lecture (ou en écriture) par l'intermédiaire de l'option `xr` (ou `xs`). Ainsi, même s'il est déclaré comme variable globale, la portée d'un canal peut être limitée à deux processus (le lecteur et l'écrivain exclusifs). Cette restriction permet de déceler des erreurs de spécification du modèle, et facilite la mise en œuvre de la réduction d'ordre partiel, utilisée pour réduire l'espace de recherche [HP94] lors de la vérification de propriétés LTL.

3.1.3 Les processus

Un processus est défini par ses éventuels paramètres et variables locales et un ensemble d'instructions. Par défaut un processus n'est pas instancié. Il peut l'être par l'emploi de l'instruction `run` ou s'il est déclaré comme processus initialement actif par l'emploi de l'option `active[N]`. Le langage Promela prévoit l'emploi d'un processus principal (`init`) chargé, si besoin est, d'« activer » le système. Ce processus un peu particulier est unique et est considéré comme initialement actif. Il n'est pas nécessaire de déclarer le processus `init` si d'autres processus du système sont déclarés comme initialement actifs.

Exemple 3.2 (Déclaration et instanciation de processus) *Considérons le code Promela suivant :*

```
Message Table[5];

proctype Sender(chan in, chan out) {
xr in;
xs out;
...
}
proctype Receiver(chan in, chan out) {
xr in;
xs out;
...
}

active[1] proctype Watchdog() {
...
}

init {
chan StoR = [3] of {Message};
chan RtoS = [0] of {Acknowledgement};
run Sender(RtoS,StoR);
run Receiver(StoR,RtoS);
}
```

Le code ci-dessus décrit le squelette d'un protocole de communication. L'émetteur (`Sender`) et le récepteur (`Receiver`) sont déclarés comme des processus paramétrés par deux canaux d'entrée (`in`) et de sortie (`out`). Ils possèdent chacun l'exclusivité en lecture (resp. écriture) sur leur canal d'entrée (resp. sortie) spécifiée par l'option `xr` (resp. `xs`). A l'état initial du système seuls deux processus ont été instanciés : `init` qui l'est par défaut et `Watchdog` qui est déclaré comme actif. Le processus `init` instancie deux processus (`Sender` et `Receiver`) en leur passant en argument les deux canaux `StoR` et `RtoS`.

3.2 Les instructions

Dans cette section nous décrivons les instructions du langage Promela. Elles sont invoquées au sein des processus et sont séparées par un point-virgule. Les instructions se divisent en deux catégories : les instructions élémentaires (gardées et étiquetées) et les délimiteurs de blocs. Ces derniers structurent les instructions (élémentaires ou non) en blocs et les exécutent entre elles et restreignent leur exécution.

3.2.1 Les instructions élémentaires

Une instruction comprend deux parties : la garde et l'action. L'exécution d'une instruction se résume à la structure conditionnelle : « si la garde est satisfaite alors exécuter l'action, sinon attendre que la garde devienne satisfaite ». Quelle que soit l'instruction Promela, la garde porte au moins sur une valeur du compteur ordinal du processus et l'action modifie celle-ci. Dans cette section nous détaillons d'abord l'ensemble des expressions (booléennes ou entières) de Promela puis l'ensemble des instructions élémentaires Promela.

Expressions

La majorité des instructions élémentaires de Promela font intervenir des expressions arithmétiques ou booléennes. Leur syntaxe et leur sémantique sont identiques à celles du langage C auxquels ont été ajoutés quelques opérateurs ou opérandes propres à Promela :
- Des opérateurs unaires permettant d'obtenir des informations sur un canal `q` (vide, plein, nombre d'éléments contenus) : `empty(q)`, `full(q)`, et `len(q)`.
- Les expressions conditionnelles ont le même comportement que celles dont dispose le langage C, mais leur syntaxe est différente : `(e1 -> e2 : e3)` signifie « l'expression `e2` » si l'expression booléenne `e1` est vraie, `e3` sinon.
- Des opérandes correspondant à des variables globales non modifiables par l'utilisateur :
  - `_pid`, qui donne l'identifiant (sur un octet) du processus en cours.
  - `_last`, l'identifiant du dernier processus activé.
  - `timeout`, une variable prédéfinie booléenne qui est vraie lorsqu'aucune instruction du système n'est exécutable.
- Une fonction d'instanciation dynamique de processus `run proc(params)` qui renvoie vrai si le processus (`proc`) a pu être instancié. L'instanciation proprement dite du processus `proc` est un effet de bord de la fonction `run`. Si le nombre de processus courants excède 255, le résultat de la fonction `run` est faux (et aucun processus supplémentaire n'a été instancié). En raison des effets de bord, `run` est généralement utilisée sous la forme d'une instruction gardée (`run proc(params);`), on ne peut en outre l'employer deux fois dans une même expression.

Gardes

On appelle « garde » une instruction dont l'« action » ne porte que sur l'évolution du compteur ordinal mais qui ne peut être exécutée que si une expression booléene est satisfaite. Une telle instruction s'écrit sous la forme `expression;`. Une garde peut interrompre l'exécution d'un processus ou être employée par certains délimiteurs de blocs comme les branchements multivoies (voir section 3.2.2).

Affectations

Les affectations en Promela s'écrivent suivant la syntaxe du langage C `var = expression`. S'il est possible de ranger dans des tableaux des valeurs de type structuré, l'affectation d'une case devra se faire champ par champ.

Communication à travers les canaux

La gestion des canaux se fait selon la politique FIFO ou « premier trouvé » (dans le cas de communications « signées »), si la taille du canal est non nulle, ou par *rendez-vous*, dans le cas contraire. L'instruction `namechan !expr` écrit la valeur de l'expression `expr` dans le canal `namechan`. De même, la commande `namechan ?var` retire le plus ancien élément de la file et l'affecte à la variable `var`. A l'instar des affectations, la lecture et l'écriture de variables scalaires de type structuré sont autorisées (champ par champ).

Il est aussi possible de transmettre une liste de valeurs ou de spécifier le type du message en lui associant une signature :

Transmission	Ecriture	Lecture
Une valeur	`name !e1`	`name ?v1`
Une liste	`name !e1, e2, e3...`	`name ?v1, v2, v3...`
Valeur signée	`name !e1 (const1, e2)`	`name ?v(const2, eval(e3))`

TAB. 3.2 Transmissions de messages *via* un canal

Dans le tableau 3.2, on associe, sur chaque ligne, l'écriture (à gauche) à la lecture (à droite) de valeurs dans le canal `name`. Sur la première ligne, la valeur de l'expression `e1` sera insérée en fin de FIFO, et la variable `v1` prendra la première valeur rangée dans la FIFO. Sur la deuxième ligne plusieurs expressions (`e1`, `e2`, `e3`...) seront insérées et plusieurs variables (`v1`, `v2`, `v3`...) seront affectées : `v1` prendra la valeur de `e1`, `v2` celle de `e2` etc. La troisième ligne est plus compliquée : Une seule variable (`v`) sera affectée par l'expression `e1` mais à la condition *sine qua non* que les signatures d'écriture `(const1, e2)` et de lecture `(const2, eval(e3))` soient identiques. Autrement dit que les constantes `const1` et `const2` soient les mêmes et que la valeur de l'expression `e3` soit égale à la valeur obtenue en évaluant, lors de l'écriture, l'expression `e2`. On parle alors de communication *signée*.

Le langage Promela propose d'autre politiques d'accès au canaux que nous n'utiliserons pas :

- la politique de lecture en mode « premier trouvé » pour les communications signées,
- la politique de lecture sans consommation,
- la possibilité ou non d'exécuter une instruction de lecture/écriture suivant n'importe quel mode.

Enfin, toute communication à travers un canal constitue une garde : une instruction de lecture (resp. écriture) ne pourra être exécutée si le canal est vide (resp. plein). De même, lors d'une lecture signée suivant la politique FIFO, si la signature du premier élément ne respecte pas la signature spécifiée, l'instruction deviendra bloquante. Dans le cas d'une politique « premier trouvé », le phénomène se produira si aucun des éléments présents dans le canal ne respecte la signature demandée.

3.2.2 Les délimiteurs de blocs

Sélections, boucles et branchements

Le langage Promela propose aussi des structures de contrôle qui diffèrent de celles du langage C. La sélection peut être indéterministe, les options sont précédées par le symbole `::`, et les délimiteurs sont `if` et `fi` :

```
if
::  option 1
::  ...
::  option n
fi;
```

Les options sont constituées d'une ou plusieurs instructions séquentielles, et ne sont pas nécessairement exclusives ce qui permet de simuler des comportements indéterministes : chaque option définissant une exécution *possible* du système, leur probabilités respectives étant ignorées, même les séquences les plus improbables sont envisagées lors de la vérification.

Une option peut être envisagée si la première instruction qu'elle contient est exécutable. Dans le cas d'un ensemble d'instructions, seule la première instruction de la séquence doit être exécutable. Une séquence peut commencer par l'instruction `else` qui n'est exécutée que pour les cas où *aucune* autre option n'est exécutable. Une instruction `else` peut être invoquée plusieurs fois au sein d'une même sélection, leur (éventuelle) activation sera indéterministe au même titre que des gardes non exclusives. En revanche l'instruction `else` est incompatible avec les gardes sensibles à la variable `timeout`.

```
if
::(i>1)  -> ...
::else   -> ...
::else   -> ...
fi;
```

```
if
::(i>1)    -> ...
::timeout  -> ...
::timeout  -> ...
fi;
```

```
if
::(i>1)    -> ...
::else     -> ...
::timeout  -> ...
fi;
```

Observons les trois sélections ci-dessus, elles font toutes intervenir trois gardes, la première (`i>1`) est insensible à `timeout`. Considérons maintenant les deux dernières gardes de ces trois sélections :

- Dans la première sélection les deux dernières gardes ne sont que des instructions `else`.
- Dans la deuxième sélection ces gardes invoquent cette fois la variable `timeout`.
- Dans la troisième sélection, on recense une instruction `else` *et* une garde sensible à `timeout`. Cette dernière sélection est syntaxiquement invalide.

Ajoutons qu'il est possible qu'aucune option ne soit exécutable comme dans les deux cas ci-dessous :

```
if
::(i>0) -> ...
::(i<0) -> ...
fi;
```

```
if
::(i>0)   ->  ...
::(i<0)   ->  ...
::timeout ->  ...
fi;
```

Dans le premier cas, le délimiteur de bloc `if ...fi;` devient bloquant si `i` est nul, le processus ne pourra reprendre son exécution que lorsqu'un autre processus aura modifié `i`. Dans le second cas, le processus reprendra son exécution si un autre processus a modifié `i` ou si tous les autres processus du système sont eux aussi bloqués. Ajoutons qu'une sélection qui comporte au moins une option « `else` » ne peut en aucun cas être bloquante.

A la sélection s'ajoutent les boucles qui emploient la même syntaxe encadrée cette fois par les mots clefs `do` et `od`. La fin d'une option se traduit par un retour au début de la boucle à moins que l'instruction `break` ne soit appelée (dans ce cas, il y a branchement sur la première instruction suivant la boucle).

```
i =0;
do
::(i <254) -> i ++;
::(1)       -> break;
od;
```

Le code ci-dessus affecte à la variable `i` une valeur aléatoire comprise entre 0 et 255 : Elle est tout d'abord initialisée à 0, puis, tant que `i` est inférieur à 254, la boucle permet soit de l'incrémenter (`i++`), soit d'interrompre le procédé (`break`). Si `i` atteint sa valeur maximale (255), seule la garde `(1)` peut être exécutée et la boucle s'achève (`break`).

Enfin, Promela autorise les branchements caractérisés par des étiquettes (`LabelName :`) et des instructions de saut (`goto LabelName`).

Atomicité

Il est possible de regrouper l'exécution d'un ensemble séquentiel d'instructions élémentaires déterministes en un seul pas d'exécution. On utilise pour cela le délimiteur `d_step{instructions}`. Dans le cas d'instructions indéterministes, le délimiteur `atomic` remplit un rôle similaire : la séquence est amorcée lorsque la première instruction est exécutable, si l'une des instructions qui la suivent n'est pas exécutable (on dit de l'instruction qu'elle est "*bloquante*"), il y a perte d'atomicité et les autres processus du système peuvent être normalement activés. La séquence atomique ne pourra reprendre que lorsque l'instruction bloquante sera à nouveau exécutable. Cette reprise se fait elle aussi de façon atomique. Elle n'est ni immédiate (il n'y a aucune priorité de la séquence sur l'exécution des autres processus) ni systématique (l'instruction bloquante peut le rester indéfiniment).

3.3 Le Langage DPN

Disydent Process Network est une bibliothèque pour simuler des réseaux de processus Kahn [Kah74] qui utilise la bibliothèque POSIX. Ce langage permet d'implémenter des processus concurrents et des canaux de communication dont l'accès est géré par la politique FIFO. La programmation en DPN reprend le synopsis (fourni par le manuel DPN) ci-dessous :

```
int main()
{
MainTask mt;
// Création des objets Canaux
...
// Création des objets Processus
...
```

```
// Initialisation des objets Processus
...
// Démarage des objets Processus en tâche de fond
...
// Attente de la fin de la simulation
mt.start()
}
```

La création d'objet « canaux » mentionne explicitement le processus producteur et le processus consommateur, la profondeur du canal, ainsi que la taille (en octets) des données :

```
Channel(Task* producer, Task* consumer, int nword, int wsize);
```

La définition du comportement d'un processus est opérée par la surcharge de la fonction membre `run()` de la classe `Task` :

```
class monProcessus : public Task {
    public:
        monProcessus (char* name) : Task(name) {}
        void *run()
            {
            ...
            }
    ...
}
...
monProcessus p("proc");
...
```

Les communications à travers des canaux sont réalisées à l'aide des méthodes suivantes :

```
int channelRead(Channel* channel, void* buf, unsigned long nword);
int channelWrite(Channel* channel, void* buf, unsigned long nword);
```

Le langage Promela permet de simuler ces éléments à l'aide de primitives plus abstraites.

3.4 Sous-ensemble Promela-DPN

Nous avons choisi d'orienter nos travaux autour du langage Promela car celui-ci permet de simuler des applications DPN et nous offre un cadre de comparaison avec d'autres techniques de model-checking (SPIN en particulier). Cela dit, notre objectif n'est pas de simuler Promela. Dans cette section nous décrivons le sous-ensemble de Promela que nous avons choisi pour simuler des applications DPN dans le but de construire un model-checker.

3.4.1 Réseau statique de processus

L'initialisation des tâches d'un système DPN est effectuée à l'aide d'une fonction principale qui procède en trois phases :

1. Création des Tâches et des Canaux.
2. Etablissement des liens entre Tâches et Canaux.
3. Activation des Tâches et simulation.

Ces trois étapes peuvent être simulées en Promela à l'aide du processus `init` en procédant à une initialisation *atomique* des canaux puis des processus. La portée de chaque canal est établie en paramétrant chaque processus avec les canaux dans lesquels ils lisent ou écrivent des informations. Ce traitement fait intervenir l'instruction `run` de Promela qui autorise, en outre, l'instanciation *dynamique* de processus. Cette dernière fonctionnalité ne présente aucun intérêt pour la simulation d'application DPN si ce n'est pour l'instanciation, *initiale et définitive*, des tâches. Dans le cas qui nous préoccupe, c'est-à-dire sans instanciation dynamique de processus, le mode d'instanciation utilisant le mot-clef `active` est suffisant. De plus, l'instanciation dynamique complique la représentation symbolique d'ensemble d'états.

Aussi, le sous-ensemble de Promela dont nous nous sommes dotés n'autorise que les instanciations uniques de processus actifs. On simule n instanciations d'un même processus par n processus distincts :

```
active[N] proctype p() {
...
}
```

devient

```
active[1] proctype p_1(){...}
active[1] proctype p_2(){...}
...
active[1] proctype p_N(){...}
```

L'établissement des liens entre processus et canaux ne sera pas nécessaire si ces derniers sont déclarés comme des variables globales. L'emploi de canaux multi-lecteurs ou multi-écrivains peut être interdit par l'emploi des options `xr` et `xs`.

Exemple 3.3 (Passage de DPN à Promela) *Considérons le code DPN suivant :*

```
int main()
{
MainTask mt;
// Création des Tâches :
Task* t1("proc1");
Task* t2("proc2");
...
// Création des Canaux :
Channel *c1(t1, t2, 3, 1);
Channel *c2(t2, t1, 2, 4);
...
}
```

Le synopsis ci-dessus décrit la création de deux tâches `t1` *et* `t2` *auxquelles on a associé les noms « proc1 » et « proc2 ». Deux canaux* `c1` *et* `c2` *sont créés :* `c1` *(resp.* `c2`*) a* `t1` *(resp.* `t2`*) comme producteur et* `t2` *(resp.* `t1`*) comme consommateur, il peut contenir 3 (resp. 2) messages de 1 octet (resp. de 4 octets).*

Dans notre sous-ensemble de Promela, cette déclaration est effectuée de la manière suivante :

```
chan c1 = [3] of {byte};
chan c2 = [2] of {int};

active[1] proctype proc1 () {
xs c1;
xr c2;
}

active[1] proctype proc2 () {
xs c2;
xr c1;
}
```

3.4.2 Communications entre processus

Le langage Promela permet la communication entre les processus à l'aide de canaux (FIFO ou *rendez-vous*) et de variables partagées. Le langage DPN n'autorise que les communications point à point au travers de canaux FIFO de valeurs non signées (voir section 3.2.1). En effet, si plusieurs tâches implantées physiquement accèdent à une variable partagée, l'exclusivité ne peut être garantie. Ce problème ne se pose pas avec Promela où chaque instruction est exécutée de façon atomique. Ce problème se retrouve aussi dans l'emploi des canaux multilecteurs ou multiécrivains pour lesquels DPN ne garantit aucune exclusivité (en lecture ou en écriture) ou pour les communications par rendez-vous qui n'appartiennent pas à la sémantique DPN.

Aussi le sous-ensemble de Promela ne devrait autoriser que les communications par FIFO non signées. Néanmoins l'utilisation de variables partagées nous est apparue utile pour deux raisons :

- Se donner des éléments de comparaison avec d'autres outils lors de l'étude de systèmes pour lesquels l'exclusivité d'accès aux variables est supposée acquise.
- Modéliser des comportements *extérieurs* au système à étudier, comme l'environnement dans lequel il évolue, ou conserver des informations utiles à la vérification de propriétés temporelles.

Nous avons choisi de conserver la possibilité d'utiliser des variables partagées et de ne les écarter que si leur usage était susceptible d'amoindrir les performances du model-checker à construire. Bien entendu, l'implémentation des composants DPN dans ce sous-ensemble de Promela ne devra en aucun cas comporter de telles variables.

3.4.3 Instructions élémentaires

Toutes les instructions élémentaires de Promela simulent des instructions DPN :

1. Les gardes sont nécessaires pour l'évaluation des branchements (`if`).
2. Les expression Promela (arithmétiques ou booléennes) sont pratiquement identiques à celles de DPN qui sont les mêmes qu'en langage C. La différence réside dans les opérateurs et opérandes propres à Promela :
   - Les opérateurs unaires donnant des informations sur les canaux (`full`, `empty` et `len`) n'ont pas d'équivalent en DPN. Ils ne seront pas utilisés pour les modèles d'applications DPN,

mais peuvent s'avérer utiles pour l'expression de propriétés temporelles. On les conservera au même titre que les variables partagées.
   - Les variables globales `_pid`, `_last` et `timeout`, n'ont pas non plus d'équivalent DPN et ont été écartées. Néanmoins l'événement `timeout` est prévu dans DPN (bien qu'il ne soit pas implémenté) et est souvent utilisé sur des systèmes « académiques » pour la comparaison de performances. Il sera donc pris en compte dans le sous-ensemble de Promela que nous utiliserons, mais sous la forme d'une instruction qui ne peut être franchie que si toutes les autres instructions du systèmes sont bloquées.
3. Les affectations ont une syntaxe et une sémantique identiques.
4. Les mécanismes de lecture/écriture non signées, dans des canaux à un seul lecteur et un seul écrivain, ont une sémantique voisine de ceux des canaux DPN.

Aussi toutes les instructions élémentaires de Promela sont conservées dans le sous-ensemble de Promela choisi.

3.4.4 Délimiteurs de Blocs

Sélections, boucles et branchements

Le langage DPN prévoit l'utilisation des structures de contrôle conditionnelles et de répétition classiques du langage C. Ces dernières se traduisent en Promela à l'aide des délimiteurs `if` et `do` :

```
if (condition)
        instruction1;
else
        instruction2;
```

devient

```
if
::(condition) -> instruction1;
:: else -> instruction2;
fi;
```

Notons que dépourvue d'option débutée par `else` une sélection Promela peut devenir bloquante ce qui n'est pas le cas de la structure de contrôle conditionnelle de DPN. Ainsi le passage de DPN vers Promela implique l'utilisation systématique de l'instruction `else` même si celle-ci n'est suivie d'aucune instruction.

```
while (condition)
        instruction1;
```

devient

```
do
::(condition) -> instruction1;
:: else -> break;
od;
```

Les autres types de sélections (comme `switch`) ou de boucles (comme `for`) peuvent être simulées avec ces délimiteurs de blocs. D'autre part, il a été choisi de conserver la possibilité d'utiliser un nombre quelconque d'options non exclusives pour introduire de l'indéterminisme des éléments extérieurs au système.

Enfin, les étiquettes et les sauts ne sont pas autorisés dans DPN mais les traitements qu'ils réalisent (modification du compteur ordinal du processus) sont communs à toutes les instructions élémentaires de Promela. Leur conservation n'aura donc aucune influence sur un model-checker. Comme les sauts sont souvent utilisés dans la définition de systèmes « académiques », ils ont été maintenus dans le seul but de comparer les performances des méthodes mises en œuvre.

Atomicité

Les deux délimiteurs de blocs `atomic` et `d_step` n'ont pas d'équivalent DPN mais peuvent, eux aussi, être utilisés pour modéliser les éléments extérieurs au système. De plus, si l'on observe la traduction des branchements DPN vers Promela, il apparaît un état intermédiaire entre la garde testant la condition et l'instruction proprement dite. Si l'on souhaite contrôler précisément le nombre d'états générés, l'utilisation des séquences atomiques devient nécessaire. Ceci est aussi valable pour la vérification de propriétés CTL dont l'opérateur $\mathbf{EX}$ s'évalue « à la transition (et donc à l'instruction) près ».

3.5 Conclusion

Le sous-ensemble de Promela choisi permet de simuler des applications DPN mais aussi l'essentiel des systèmes « académiques » pour profiter d'un cadre de comparaison avec les autres techniques de model-checking. Les caractéristiques principales de ce sous-ensemble sont les suivantes :

- le réseau de processus est statique,
- les canaux ne peuvent qu'adopter la politique FIFO,
- les variables partagées sont conservées à condition de ne pas intervenir dans le modèle simulant les applications DPN et de ne pas nuire à l'efficacité du model-checker,
- les délimiteurs d'atomicité sont conservés.

Au sujet de ce dernier point, nous verrons au chapitre 5 que l'appréhension, par notre model-checker, de variables partagées ne diffère pas de celle des variables locales. Ceci nous a permis de mesurer notre prototype à d'autre techniques de model-checking pour des systèmes « académiques » mettant en jeu des variables partagées.

Deuxième partie

Développement - Contributions

Chapitre 4

DDD ordonnés

Dans le cadre de cette thèse, les DDD sont utilisés pour représenter des ensembles finis d'états. Nous avons vu au chapitre 2 que les DDD ont été définis comme une structure pour la représentation de *séquences d'affectations*. Les DDD représentant les séquences ne garantissent pas la canonicité de représentation de l'état résultant de séquences d'affectations. Par exemple, les séquences $a = 0; b = 0;$ et $b = 0; a = 0;$ correspondent à un unique état $(a = 0) \wedge (b = 0)$ et leurs représentation sur DDD ($a \xrightarrow{0} b \xrightarrow{0} 1$ et $b \xrightarrow{0} a \xrightarrow{0} 1$) ne sont pas équivalentes et leur union mène à un DDD non défini. Ce problème peut être résolu en imposant un ordre des affectations. L'objectif des sections suivantes est de munir les DDD d'un ordre qui tolère les affectations multiples. Cet ordre est adapté à la définition inductive des opérateurs sur les DDD (opération ensemblistes et homomorphismes inductifs). Le principe de construction que nous proposons est basé sur la représentation des files défini par [CEPA+02]. Nous montrons ici qu'il garantit les propriétés de canonicité et de bonne définition.

Sommaire

4.1 Définitions

Les définitions qui suivent permettent de définir un environnement de construction des DDD assurant leur canonicité et leur « bonne définition » lorsqu'ils représentent des ensembles d'états.

Un état est une conjonction d'associations (v, x) où v est une variable d'état et $x \in Dom(v)$. On appellera « configuration » cette association. L'ordre d'apparition des associations dans la conjonction n'importe pas (la conjonction est commutative) ; il faut donc fixer un ordre d'apparition pour que la représentation soit canonique. On commence par définir les catégories de variables qui sont utilisées.

Définition 4.1 (Variables scalaires et conteneurs) *On distingue deux sous-ensembles disjoints et complémentaires de E, l'ensemble des variables scalaires S et l'ensemble des conteneurs C : $E = S \cup C$ et $S \cap C = \emptyset$.*

- *Une variable $e \in S$ est dite scalaire lorsque sa grandeur est exprimable à l'aide d'une valeur unique dans $Dom(e)$. Par défaut, les variables d'un DDD sont scalaires. Les variables vectorielles (tableaux et structures) sont assimilées à un ensemble de variables scalaires de taille fixe.*
- *Une variable $e \in C$ est un conteneur si sa grandeur est définie par un ensemble ordonné de valeurs.*

Les variables conteneurs ont leur domaine de définition étendu avec une valeur supplémentaire (notée $\sharp$) désignant un délimiteur de fin de bloc.

Définition 4.2 (Configuration d'une variable) *Pour une variable (e) donnée il existe une unique représentation de l'affectation $e = x$.*
- *La configuration d'une variable scalaire e et d'une valeur x est représentée sur le DDD $e \xrightarrow{x} 1$.*
- *La configuration d'un conteneur c et d'une liste $\{c_1 \ldots c_n\}$ est représentée par le DDD $c \xrightarrow{c_1} c \to \ldots c \xrightarrow{c_n} c \xrightarrow{\sharp} 1$ qui peut s'écrire sous la forme de DDD concaténés : $(c \xrightarrow{c_1} 1) \cdot (\ldots) \cdot (c \xrightarrow{c_n} 1) \cdot c \xrightarrow{\sharp} 1 = \bullet_{i=1}^{n}(c \xrightarrow{c_i} 1) \cdot c \xrightarrow{\sharp} 1$.*

Dans le cas d'un conteneur, les valeurs c_n sont rangées dans l'ordre de la liste, chaque séquence devant se terminer par $c \xrightarrow{\sharp} 1$.

Il en existe une infnité d'autres, mais nous avons choisi celui-ci pour sa lisibilité.

Un chemin représentant un état est une succession de strates. Chaque strate correspond à une variable. Les variables scalaires occupent des strates de profondeur 1 tandis que les conteneurs occupent des strates de taille variable.

Pour garantir la canonicité et la « bonne définition » des DDD représentant des états, nous imposons un ordre unique d'apparition des strates le long des chemins :

Définition 4.3 (Ordre de variables) *Un ordre de variables $\mathcal{O}$ est défini inductivement comme suit :*
- *$\mathcal{O} = \emptyset$ ou*
- *$\mathcal{O} = (e, \mathcal{O}')$ où $e \in E$ et $\mathcal{O}'$ est un ordre de variables.*

Dans le cas dégénéré d'un ordre à une variable, on note l'ordre $(e, \emptyset)$ par le singleton (e).

Définition 4.4 (Egalités entre deux ordres) *Deux ordres $\mathcal{O}_1$ et $\mathcal{O}_2$ sont égaux (on note $\mathcal{O}_1 = \mathcal{O}_2$) si et seulement si :*
- *$\mathcal{O}_1 = \emptyset$ et $\mathcal{O}_2 = \emptyset$ ou*
- *$\mathcal{O}_1 = (e_1, \mathcal{O}'_1)$ et $\mathcal{O}_2 = (e_2, \mathcal{O}'_2)$ avec $e_1 = e_2$ et $\mathcal{O}'_1 = \mathcal{O}'_2$.*

Définition 4.5 (Concaténation d'ordres) *Soient $\mathcal{O}_1$ et $\mathcal{O}_2$ deux ordres sur E, la concaténation $\mathcal{O} = \mathcal{O}_1 \cdot \mathcal{O}_2$ est définie inductivement comme suit :*

$$\mathcal{O} = \mathcal{O}_1 \cdot \mathcal{O}_2 = \begin{cases} \mathcal{O}_2 & \text{si } \mathcal{O}_1 = \emptyset \\ (e, \mathcal{O}'_1 \cdot \mathcal{O}_2) & \text{si } \mathcal{O}_1 = (e, \mathcal{O}'_1) \end{cases}$$

Il nous reste à associer un ordre à la construction d'un DDD :

Définition 4.6 (DDD Ordonné) *Soient d un DDD bien défini et $\mathcal{O}$ un ordre de variables. On dit que d respecte l'ordre $\mathcal{O}$ si et seulement si :*

- $d \equiv 0$ *ou*
- $d = 1$ *et* $\mathcal{O} = \emptyset$ *ou*
- $d = (e, \alpha) \not\equiv 0$ *et* $\mathcal{O} = (e, \mathcal{O}')$
  - $e \in S$, $\forall x \in Dom(e)$, $\alpha(x)$ *est bien défini et respecte l'ordre* $\mathcal{O}'$.
  - $e \in C$, $\forall x \in Dom(e)$, $\alpha(x)$ *est bien défini et respecte l'ordre* $\mathcal{O}$ *et* $\alpha(\sharp)$ *respecte l'ordre* $\mathcal{O}'$.

La valeur $\sharp$ *est ici une valeur n'appartenant pas au domaine de* E*, pour plus de simplicité, on choisira une valeur arbitraire n'appartenant à aucun domaine des variables de* E *:* $\forall e \in E, \sharp \notin Dom(e)$.

On dit que deux DDD sont de même classe (ou *compatibles*) s'ils respectent le même ordre.

4.2 Propriétés garanties

Au chapitre 2, nous avons noté que le résultat des opérations ensemblistes sur deux DDD bien définis n'est pas nécessairement bien défini. La compatibilité d'ordre entre DDD permet d'assurer la bonne définition d'un DDD construit à partir des opérations ensemblistes.

Proposition 4.7 (Ordres et Opérations ensemblistes) *Soient deux DDD* d_1 *et* d_2*, bien définis et de même ordre* $\mathcal{O}$*. Le résultat de l'application de* $d_1 \diamond d_2, (\diamond \in \{+, \times, \setminus\})$ *est bien défini et d'ordre* $\mathcal{O}$.
Démonstration *: La démonstration se fait par récurrence.* d_1 *(resp.* d_2*) est bien défini, on ignore donc les cas où* $d_1 = \top$ *(resp.* $d_2 = \top$*).*

- $d_1 \equiv 0$ *(resp.* $d_2 \equiv 0$*), comme* d_1 *et* d_2 *sont supposés être de même ordre, le résultat de* $d_1 \diamond d_2$ *vaut soit* 0 *soit* d_2 *(resp.* d_1*) qui sont bien définis et respectent tous les deux l'ordre* $\mathcal{O}$.
- $d_1 = 1$*, alors* $\mathcal{O} = \emptyset$*, donc* $d_2 \in \{0, 1\}$ *(*0 *et* 1 *sont les seuls DDD respectant l'ordre* $\emptyset$*) et* $d_1 \diamond d_2 \in \{0, 1\}$ *qui est bien défini et respecte l'ordre* $\mathcal{O} = \emptyset$.
- $d_1 = (e, \alpha_1) \not\equiv 0$*, alors,* $\mathcal{O} = (e, \mathcal{O}')$*, le cas* $d_2 \equiv 0$ *ayant déjà été traité,* $d_1 \diamond d_2 = (e, \alpha_1 \diamond \alpha_2)$ *respecte l'ordre* $\mathcal{O}$ *si :*
  - $e \in S$ *et* $\forall x \in Dom(e)$*, et* $\alpha_i(x)$ *respecte l'ordre* $\mathcal{O}'$ *ou*
  - $e \in C$ *et* $\forall x \in Dom(e)$*, et* $\alpha_i(x)$ *respecte l'ordre* $\mathcal{O}$ *et* $\alpha_i(\sharp)$ *respecte l'ordre* $\mathcal{O}'$.

On remarque que les deux dernières assertions sont vérifiées du fait de la définition des DDD ordonnés (4.6), la propriété étant vérifiée au niveau des feuilles (1 *ou* 0*), et par récurrence sur chacun des nœuds du résultat* $d_1 \diamond d_2$. □

Enfin, on vérifie que la concaténation de DDD, dans ce cadre de construction, est compatible avec la concaténation d'ordres :

Proposition 4.8 (Ordres et Concaténation) *Soient deux DDD* d_1 *et* d_2 *d'ordres respectifs* $\mathcal{O}_1$ *et* $\mathcal{O}_2$*, et bien définis, alors* $d = d_1 \cdot d_2$ *est bien défini et d'ordre* $\mathcal{O} = \mathcal{O}_1 \cdot \mathcal{O}_2$.
Démonstration *: En annexe page 127.* □

Les définitions 4.2, 4.6 et la proposition 4.8 mènent à la propriété suivante :

Proposition 4.9 (Canonicité) *Soit* $\mathcal{O}$ *un ordre, et* $\mathcal{S}$ *une conjonction de variables (état), alors la représentation de* $\mathcal{S}$ *sur un DDD respectant l'ordre* $\mathcal{O}$ *est unique.*
Démonstration *: Elle se fait par récurrence :*

- *si* $\mathcal{O} = \emptyset$ *le seul DDD n'étant pas équivalent à* 0 *respectant l'ordre* $\mathcal{O}$ *est le DDD unique* 1 *qui correspond à la configuration vide* $\mathcal{S} = \emptyset$.

– *si* $\mathcal{O} = (e, \mathcal{O}') = (e) \cdot \mathcal{O}'$ *alors* $\mathcal{S}$ *peut s'écrire* $\mathcal{S} = (e, x) \cdot \mathcal{S}'$ *et la représentation de* $\mathcal{S}$ *est unique si la représentation de* $\mathcal{S}'$ *respectant* $\mathcal{O}'$ *l'est aussi.*

La définition 4.2 garantit la canonicité de la représentation de la configuration (e, x)*, et par récurrence, la canonicité de toute représentation de séquence sur un DDD pourvu d'un ordre contenant toutes les variables affectées.* □

L'opérateur de concaténation, permet de construire un unique DDD représentant un état si l'on dispose d'un ordre sur l'ensemble des variables de E. On illustre cette propriété par l'exemple suivant :

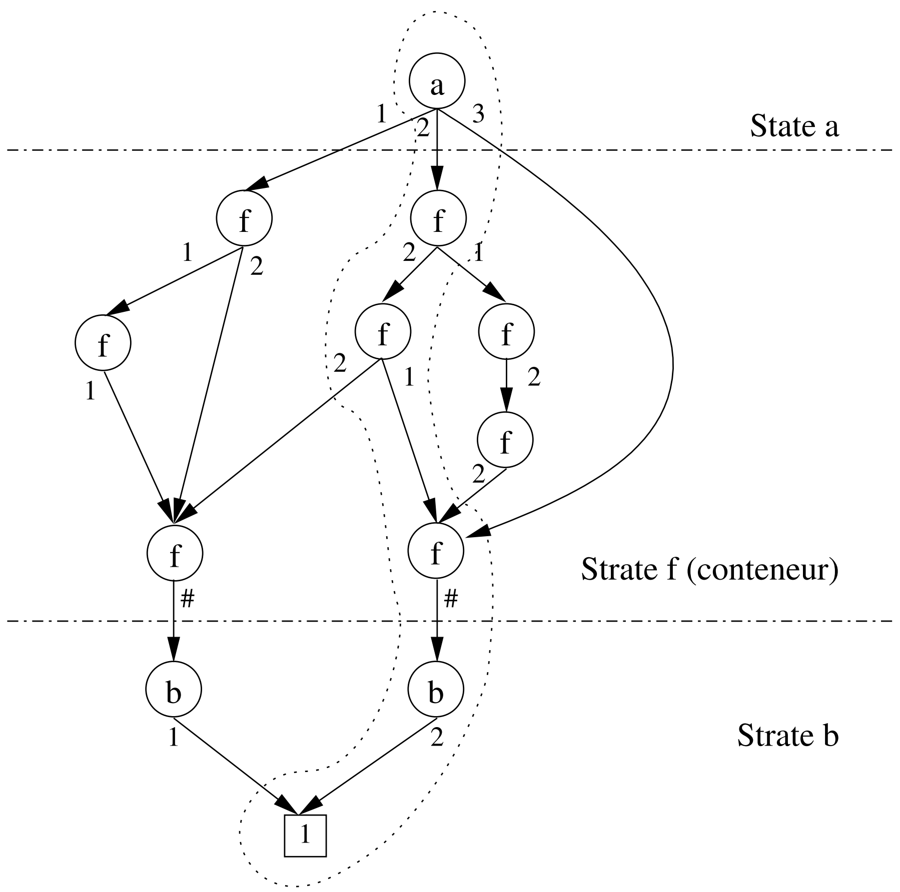


FIG. 4.1 Séquences d'affectations respectant l'ordre (a,f,b) où f est un conteneur

Exemple 4.1 *La figure 4.1 donne le DDD résultant de l'union des 6 DDD suivants :*

$$d_1 = a \xrightarrow{1} f \xrightarrow{1} f \xrightarrow{1} f \xrightarrow{\sharp} b \xrightarrow{1} 1 \quad = \quad (a \xrightarrow{1} 1) \cdot (f \xrightarrow{1} f \xrightarrow{1} f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{1} 1) \cdot 1$$

$$d_2 = a \xrightarrow{1} f \xrightarrow{2} f \xrightarrow{\sharp} b \xrightarrow{1} 1 \quad = \quad (a \xrightarrow{1} 1) \cdot (f \xrightarrow{2} f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{1} 1) \cdot 1$$

$$d_3 = a \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{\sharp} b \xrightarrow{1} 1 \quad = \quad (a \xrightarrow{2} 1) \cdot (f \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{1} 1) \cdot 1$$

$$d_4 = a \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{1} f \xrightarrow{\sharp} b \xrightarrow{2} 1 \quad = \quad (a \xrightarrow{2} 1) \cdot (f \xrightarrow{2} f \xrightarrow{1} f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{2} 1) \cdot 1$$

$$d_5 = a \xrightarrow{2} f \xrightarrow{1} f \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{\sharp} b \xrightarrow{2} 1 \quad = \quad (a \xrightarrow{2} 1) \cdot (f \xrightarrow{1} f \xrightarrow{2} f \xrightarrow{2} f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{2} 1) \cdot 1$$

$$d_6 = a \xrightarrow{3} f \xrightarrow{\sharp} b \xrightarrow{2} 1 \quad = \quad (a \xrightarrow{3} 1) \cdot (f \xrightarrow{\sharp} 1) \cdot (b \xrightarrow{2} 1) \cdot 1$$

Chaque séquence est construite par concaténation, en respectant l'ordre $\{a, f, b\}$*, des affectations individuelles sur les trois variables. Le DDD* d_4 *est entouré en pointillés. Cet exemple sera repris au chapitre 6.*

4.3 Préservation de l'ordre

Dans le cadre de cette thèse, les homomorphismes inductifs seront utilisés pour construire des ensembles d'états en modifiant certaines configurations de variables. Le DDD résultant de l'application d'un homomorphisme devra avoir un ordre compatible avec le DDD d'origine. Parmi les homomorphismes qui respectent l'ordre d'un DDD on trouve :

1. l'homomorphisme nul (le DDD 0 est compatible avec n'importe quel ordre),
2. l'homomorphisme identité,
3. tout homomorphisme inductif $\langle\ \mathsf{h}\ ()\ \rangle$ qui appliqué au couple (e, x) respecte l'encodage de la variable e :
   - si e est une variable scalaire alors $\langle\ \mathsf{h}\ ()\ \rangle(e, x) = e \xrightarrow{x' \in Dom(e)} \langle\ \mathsf{h'}\ ()\ \rangle$ où $\langle\ \mathsf{h'}\ ()\ \rangle$ respecte également l'ordre d'un DDD.
   - si e est un conteneur on peut modifier, insérer ou retirer des nœuds étiquetés par e n'altère pas la dernière configuration portant sur le le marqueur de fin de bloc $\sharp$:
     - $\langle\ \mathsf{h}\ ()\ \rangle(e, x \neq \sharp) = e \xrightarrow{x' \in Dom(e)} \langle\ \mathsf{h'}\ ()\ \rangle$,
     - $\langle\ \mathsf{h}\ ()\ \rangle(e, x \neq \sharp) = e \xrightarrow{x' \in Dom(e)} e \xrightarrow{x'' \in Dom(e)} \dots e \xrightarrow{x^{(n)} \in Dom(e)} \langle\ \mathsf{h'}\ ()\ \rangle$
     - $\langle\ \mathsf{h}\ ()\ \rangle(e, x \neq \sharp) = \langle\ \mathsf{h'}\ ()\ \rangle$.
     - $\langle\ \mathsf{h}\ ()\ \rangle(e, x = \sharp) = e \xrightarrow{\sharp} \langle\ \mathsf{h'}\ ()\ \rangle$,
     - $\langle\ \mathsf{h}\ ()\ \rangle(e, x = \sharp) = e \xrightarrow{x' \in Dom(e)} e \xrightarrow{x'' \in Dom(e)} \dots e \xrightarrow{x^{(n)} \in Dom(e)} e \xrightarrow{\sharp} \langle\ \mathsf{h'}\ ()\ \rangle$

     où $\langle\ \mathsf{h'}\ ()\ \rangle$ respecte également l'ordre d'un DDD.

Cependant, aucun homomorphisme respectant au moins une de ces règles ne peut réaliser l'affectation $a \leftarrow b$ sur le DDD de la figure 4.1 : la valeur de b sur un chemin n'est connue « qu'après » avoir traversé les strates a et f lorsque les arcs issus du nœud étiqueté par a ne sont plus modifiables. Un élément de solution à ce problème est apporté par [CEPA+02] qui définit un homomorphisme de permutation ($\langle\ \mathsf{swap}\ ()\ \rangle$) de deux variables. Nous en généralisons le fonctionnement pour construire un homomorphisme ($\langle\ \mathsf{replace}\ ()\ \rangle$) chargé de replacer, au niveau de la strate ou il a été appelé, une variable *scalaire* située dans une strate inférieure quelconque.

Au chapitre 5 nous nous appuierons sur $\langle\ \mathsf{replace}\ ()\ \rangle$ pour démontrer que certains homomorphismes respectent l'ordre des DDD bien qu'ils n'appartiennent à aucune des catégories citées plus haut.

On construit d'abord l'homomorphisme suivant :

Définition 4.10 (Homomorphisme $\langle\ \mathsf{up}\ ()\ \rangle$ [CEPA+02]) *L'homomorphisme inductif* $\langle\ \mathsf{up}\ ()\ \rangle$ *est défini inductivement comme suit :*

$$\begin{aligned} \langle\ \mathsf{up}\ (var, val)\ \rangle(1) &= \top \\ \langle\ \mathsf{up}\ (var, val)\ \rangle(e, x) &= e \xrightarrow{x} var \xrightarrow{val} \langle\ \mathsf{id}\ \rangle \end{aligned}$$

On se sert de cet homomorphisme pour faire « remonter » une variable scalaire vers la strate supérieure :

Proposition 4.11 *Soit le DDD* $d = e \xrightarrow{x} d' = (e \xrightarrow{x} 1) \cdot d'$*, bien défini, alors*

$$\langle\ \mathsf{up}\ (var, val)\ \rangle(d) = e \xrightarrow{x} var \xrightarrow{val} d' = (e \xrightarrow{x} 1) \cdot (var \xrightarrow{val} 1) \cdot d'$$

est bien défini.
Démonstration : d *est bien défini, par conséquent* d' *l'est aussi. Il s'ensuit que* $(var \xrightarrow{val} 1) \cdot d'$ *est bien défini puis que* $(e \xrightarrow{x} 1) \cdot (var \xrightarrow{val} 1) \cdot d'$ *l'est aussi.* □

Définition 4.12 (Homomorphisme $\langle\ \mathsf{replace}\ ()\ \rangle$**)** *L'homomorphisme inductif* $\langle\ \mathsf{replace}\ ()\ \rangle$ *est défini inductivement comme suit :*

$$\begin{array}{rcl} \langle\ \mathsf{replace}\ (v)\ \rangle(1) & = & \top \\ \langle\ \mathsf{replace}\ (v)\ \rangle(e, x) & = & \begin{cases} \langle\ \mathsf{up}\ (e, x)\ \rangle \circ \langle\ \mathsf{replace}\ (v)\ \rangle & \text{si } e \neq v \\ e \xrightarrow{x} \langle\ \mathsf{id}\ \rangle & \text{sinon} \end{cases} \end{array}$$

Proposition 4.13 *Soit* s *une variable scalaire* ($s \in S$) *et* $d = (v, \alpha)$ *un DDD bien défini d'ordre* $(v, (s, \mathcal{O})) = (v) \cdot (s) \cdot \mathcal{O}$*, alors* $d' = \langle\ \mathsf{replace}\ (s)\ \rangle(d)$ *est bien défini et d'ordre* $(s, (v, \mathcal{O})) = (s) \cdot (v) \cdot \mathcal{O}$*.*
Démonstration : *En annexe page 127.* □

Corollaire 4.14 *Soit* d *un DDD bien défini respectant l'ordre* $\mathcal{O}_1 \cdot (s) \cdot \mathcal{O}_2$ *où* s *est une variable scalaire, alors* $d' = \langle\ \mathsf{replace}\ (s)\ \rangle(d)$ *est bien défini et d'ordre* $(s) \cdot \mathcal{O}_1 \cdot \mathcal{O}_2 = (s, \mathcal{O}_1 \cdot \mathcal{O}_2)$*.*
Démonstration : *Elle se fait par récurrence :*
- *si* $\mathcal{O}_1 = \emptyset$ *alors le résultat est immédiat :* $d = d'$*.*
- *sinon* $\mathcal{O}_1 = (e, \mathcal{O}'_1)$*,* $d = (e, \alpha)$ *et* $d' = \langle\ \mathsf{replace}\ s\ \rangle(d)$ *respecte l'ordre* $(s) \cdot \mathcal{O}_1 \cdot \mathcal{O}_2$ *si l'homomorphisme* $\langle\ \mathsf{replace}\ s\ \rangle$ *appliqué sur un DDD respectant l'odre* $\mathcal{O}'_1 \cdot (s) \cdot \mathcal{O}_2$ *respecte l'ordre* $(s) \cdot \mathcal{O}'_1 \cdot \mathcal{O}_2 = (s, \mathcal{O}'_1 \cdot \mathcal{O}_2)$*.*

La dernière assertion est assurée par la propriété 4.13. La propriété étant vérifiée pour $\mathcal{O}_1 = \emptyset$*, elle est démontrée par récurrence pour chaque ordre* $\mathcal{O}_1$*.* □

A l'aide de l'homomorphisme $\langle\ \mathsf{replace}\ ()\ \rangle$ il est donc possible d'assurer la compatibilité d'ordre entre deux DDD quels que soient leurs ordres, pourvu que ces derniers instancient le même ensemble de variables.

Exemple 4.2 *On souhaite replacer la variable* a *sur la première strate du DDD* $b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} a \xrightarrow{1} e \xrightarrow{5} 1$*. On applique l'homomorphisme* $\langle\ \mathsf{replace}\ (a)\ \rangle$ *dont l'évaluation procède comme suit :*

$$\begin{array}{rl} & \langle\ \mathsf{replace}\ (a)\ \rangle(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} a \xrightarrow{1} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle \circ \langle\ \mathsf{replace}\ (a)\ \rangle(c \xrightarrow{3} d \xrightarrow{4} a \xrightarrow{1} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle \circ \langle\ \mathsf{up}\ (c, 3)\ \rangle \circ \langle\ \mathsf{replace}\ (a)\ \rangle(d \xrightarrow{4} a \xrightarrow{1} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle \circ \langle\ \mathsf{up}\ (c, 3)\ \rangle \circ \langle\ \mathsf{up}\ (d, 4)\ \rangle \circ \langle\ \mathsf{replace}\ (a)\ \rangle(a \xrightarrow{1} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle \circ \langle\ \mathsf{up}\ (c, 3)\ \rangle \circ \langle\ \mathsf{up}\ (d, 4)\ \rangle(a \xrightarrow{1} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle \circ \langle\ \mathsf{up}\ (c, 3)\ \rangle(a \xrightarrow{1} d \xrightarrow{4} e \xrightarrow{5} 1) \\ = & \langle\ \mathsf{up}\ (b, 2)\ \rangle(a \xrightarrow{1} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\ = & a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1 \end{array}$$

4.4 Conclusion

Nous avons élaboré un cadre formel de construction des DDD, basé sur une notion d'ordre, pour assurer leur bonne définition ainsi que leur canonicité lorsqu'ils représentent des ensembles d'états. Cet ordre est préservé par les opérations ensemblistes et les homomorphismes ne modifiant ni l'ordre ni le mode d'encodage des variables. De plus si un homomorphisme repousse une variable scalaire vers une strate inférieure, il est possible, à l'aide de l'homomorphisme $\langle \mathsf{replace}\,()\,\rangle$ de rétablir l'ordre original. Nous verrons au chapitre 5 que ce cadre est suffisant pour construire un model-checker symbolique, basé sur les DDD, pour le sous-ensemble du langage Promela.

Chapitre 5

Approche Promela-DDD

Dans ce chapitre, nous montrons comment représenter, à l'aide de DDD ordonnés, un ensemble d'états Promela. Ensuite, nous montrons comment réaliser les instructions élémentaires du language Promela à l'aide d'homomorphismes inductifs et comment les articuler afin de simuler les délimiteurs de blocs. Cette étude a donné lieu à l'élaboration d'un premier prototype de model-checker symbolique basé sur les DDD. Nous présentons une étude comparative du model-checker basé sur DDD, sur BDD et du model-checker énumératif SPIN. Elle nous permet d'identifier les grandes classes de systèmes pour lesquels notre model-checker est mieux adapté que SPIN.

Sommaire

5.1 Objectifs

L'objectif de ce chapitre est d'utiliser les DDD ordonnés pour construire un model-checker symbolique pour vérifier des propriétés CTL sur des programmes écrits dans le sous-ensemble de Promela défini au chapitre 3. Nous avons vu au chapitre 2 les éléments suffisants pour construire un tel outil :

- un ensemble d'états $S_0 \subseteq S$,
- un opérateur ($select$) de sélection dans l'ensemble $S_1 \subseteq S$ du sous-ensemble des états satisfaisant une condition booléenne ϕ :

$$S_2 = select_\phi(S_1) = \{s_2 \in S_1, (M, s_2 \models \phi)\}$$

- un opérateur ($post$) qui construit l'ensemble S_2 des états successeurs à partir d'en ensemble d'états $S_1 \subseteq S$ par la relation de transition R :

$$S_2 = post(S_1) = \{s_2, \forall(s_1, s_2) \in S_1 \times S, (s_1, s_2) \in R\}$$

- un operateur (pre) qui construit l'ensemble S_1 des états prédécesseurs à partir d'en ensemble d'états $S_2 \subseteq S$ par la relation de transition R :

$$S_1 = pre(S_2) = \{s_1, \forall(s_1, s_2) \in S \times S_2, (s_1, s_2) \in R\}$$

Nous montrons d'abord comment représenter S_0 en nous basant sur la notion d'ordre du chapitre 4. Les autres opérateurs devront construire des DDD respectant le même ordre que S_0 pour garantir les propriétés de canonicité et de « bonne définition ». Nous donnons d'abord les traitements élémentaires que peuvent réaliser les instructions Promela et les opérateurs $post$ et pre qui leur sont associés. Nous montrons en annexe comment articuler ces traitements afin de simuler les délimiteurs de blocs du langage Promela.

5.2 Représentation des états d'un programme Promela

Dans cette section, nous montrons comment représenter un ensemble d'états d'un programme Promela avec des DDD ordonnés.

L'ensemble E des variables du programme Promela comprend les éléments suivants :
- les variables globales (notamment les canaux de communication)
- et pour chaque processus un compteur ordinal et les variables locales du processus.

La représentation *initiale* des éléments d'un système Promela est la suivante :
- les variables scalaires sont initialisés suivant la définition 4.2 : $var \xrightarrow{v_0} 1$ où v_0 est la valeur d'initialisation de var si celle-ci est précisée (elle est nulle par défaut),
- les canaux sont initialement vides. La première valeur du conteneur indiquera la taille du canal : $f \xrightarrow{size} f \xrightarrow{\sharp} 1$,
- les tableaux de taille fixe n sont initialisés comme n variables scalaires initialement nulles : $t_{[0]} \xrightarrow{0} t_{[1]} \xrightarrow{0} \ldots t_{[n-1]} \xrightarrow{0} 1$,
- les structures sont mises à plat et initialisées à 0 champ par champ : $S.champ_0 \xrightarrow{0} \ldots S.champ_{n-1} \xrightarrow{0} 1$,
- enfin, les processus sont eux-même le résultat de la concaténation des variables qui les composent : le processus i est encodé sous la forme $PC_i \xrightarrow{0} local_{i,0} \xrightarrow{v_{i,0}} \ldots local_{i,n-1} \xrightarrow{v_{i,n-1}} 1$ où PC_i est le compteur ordinal du processus i et où les $v_{i,j}$ sont les valeurs d'initialisation des variables locales $local_{i,j}$ si celles-ci sont spécifiées (uniquement pour les variables scalaires).

On dispose alors des configurations initiales de chaque variable d'état du programme Promela. On choisit un ordre $\mathcal{O}$ (par défaut, l'ordre d'apparition des variables dans le programme Promela) et on construit, par concaténation, la configuration initiale du programme qui respecte $\mathcal{O}$.

5.3 Représentation des instructions Promela

Toute instruction Promela est exécutée au sein d'un processus. Son exécution implique donc l'évolution du compteur ordinal (noté PC) de ce dernier. On traduit alors tout ensemble d'instructions séquentielles par un ensemble d'instructions concurrentes gardées par la valeur du compteur ordinal (notée $oldPC$) et dont les exécutions positionnent le compteur ordinal à une nouvelle valeur (notée $newPC$).

		Précondition	Modification
Garde	post	$\langle$ select $(PC = oldPC)$ $\rangle$ $\circ\langle$ select $(expr)$ $\rangle$	$\langle$ setExpr $(PC, newPC)$ $\rangle$
(`expr`)	pre	$\langle$ select $(PC = newPC)$ $\rangle$	$\langle$ setExpr $(PC, oldPC)$ $\rangle$ $\circ\langle$ select $(expr)$ $\rangle$
Affectation	post	$\langle$ select $(PC = oldPC)$ $\rangle$	$\langle$ setExpr $(var, expr)$ $\rangle$ $\circ\langle$ setExpr $(PC, newPC)$ $\rangle$
(`var=expr;`)	pre	$\langle$ select $(PC = newPC)$ $\rangle$	$\langle$ preSetExpr $(var, expr, Reached)$ $\rangle$ $\circ\langle$ setExpr $(PC, oldPC)$ $\rangle$
Ecriture dans un canal `q`	post	$\langle$ select $(PC = oldPC)$ $\rangle$ $\circ\langle$ notFull (q) $\rangle$	$\langle$ writeExpr $(q, expr)$ $\rangle$ $\circ\langle$ setExpr $(PC, newPC)$ $\rangle$
(`q!expr;`)	pre	$\langle$ select $(PC = newPC)$ $\rangle$ $\circ\langle$ notEmpty (q) $\rangle$	$\langle$ preWriteExpr $(q, expr)$ $\rangle$ $\circ\langle$ setExpr $(PC, oldPC)$ $\rangle$
Lecture dans un canal `q`	post	$\langle$ select $(PC = oldPC)$ $\rangle$ $\circ\langle$ notEmpty (q) $\rangle$	$\langle$ readVar (q, var) $\rangle$ $\circ\langle$ setExpr $(PC, newPC)$ $\rangle$
(`q?var;`)	pre	$\langle$ select $(PC = newPC)$ $\rangle$ $\circ\langle$ notFull (q) $\rangle$	$\langle$ preReadVar $(q, var, Reached)$ $\rangle$ $\circ\langle$ setExpr $(PC, oldPC)$ $\rangle$

TAB. 5.1 Traitements pour les instructions élémentaires

Le tableau 5.1 présente les traitements spécifiques associés à chaque instruction élémentaire de Promela : Pour chaque instruction et pour chaque opérateur (pre ou $post$) sont donnés les homomorphismes nécessaires pour obtenir deux ensembles d'états :

1. La colonne « Précondition » donne l'homomorphisme sélectionnant l'ensemble des états sur lesquels s'appliqueront les modifications induites par l'exécution de l'instruction.
2. La colonne « Modification » donne l'homomorphisme réalisant la modification induite par l'exécution de l'instruction sur ces derniers pour obtenir leurs successeurs ou prédécesseurs immédiats pour une instruction donnée.

Les homomorphismes du tableau 5.1 ont les comportements suivants :

- $\langle$ select (ϕ) $\rangle$ sélectionne les états satisfaisant la propriété atomique ϕ.
- $\langle$ notFull (q) $\rangle$ sélectionne les états pour lesquels le canal q est non plein.
- $\langle$ notEmpty (q) $\rangle$ sélectionne les états pour lesquels le canal q est non vide.
- $\langle$ setExpr $(var, expr)$ $\rangle$ affecte la valeur de l'expression $expr$ à la variable var.
- $\langle$ preSetExpr $(var, expr, Reached)$ $\rangle$ est l'opération réciproque de la précédente, elle ne peut être évaluée sans une connaissance de tous les états accessibles $Reached$.
- $\langle$ writeExpr $(q, expr)$ $\rangle$ insère la valeur de l'expression $expr$ dans le canal q.

– $\langle$ preWriteExpr $(q, expr)$ $\rangle$ est l'opération réciproque de la précédente.
– $\langle$ readVar (q, var) $\rangle$ lecture du premier élement de le canal q et stockage dans la variable var.
– $\langle$ preReadVar $(q, var, Reached)$ $\rangle$ est l'opération réciproque de la précédente, elle ne peut être évaluée sans une connaissance de tous les états accessibles $Reached$.

Ils seront détaillés dans la section suivante. Ces opérateurs peuvent être combinés pour simuler tous les délimiteurs de blocs Promela. Les algorithmes associés sont donnés dans l'annexe C.

5.4 Homomorphismes Spécifiques

Cette section décrit la construction des homomorphismes présentés dans le tableau 5.1. Ces homomorphismes respectent les règles de préservation de l'ordre données au chapitre 4.

5.4.1 Sélection

Nous définissons l'homomorphisme $\langle$ selectExpr $(expr)$ $\rangle$ réalisant la sélection des états satisfaisant l'expression booléenne $expr$. Cette expression peut varier suivant les valeurs prises par ses opérandes (si $e \in expr$), on note $expr_{e:=x}$ l'expression $expr$ pour laquelle l'opérande e a été remplacé par la valeur x. On note $expr = cst$ lorsque toutes les variables de l'expression sont affectées (cst désigne la valeur de l'expression).

Proposition 5.1 (Homomorphisme de Sélection) *L'homomorphisme $\langle$ SelectExpr $(expr)$ $\rangle$ décrit sélectionne les états satisfaisant l'expression booléenne $expr$.*

$$\langle \text{selectExpr } (vrai) \rangle = \langle \text{id} \rangle \tag{5.1}$$

$$\langle \text{selectExpr } (faux) \rangle = 0 \tag{5.2}$$

$$\langle \text{selectExpr } (expr) \rangle(1) = \top \tag{5.3}$$

$$\langle \text{selectExpr } (expr) \rangle(e, x) = \begin{cases} e \xrightarrow{x} \langle \text{this} \rangle & \text{si } e \notin expr \\ e \xrightarrow{x} \langle \text{selectExpr } (expr_{e:=x}) \rangle & \text{sinon} \end{cases} \tag{5.4}$$

***Explication** : Les formules 5.1 et 5.2 sont immédiates, tous les états satisfont la propriété $vrai$ et aucun ne peut satisfaire $faux$. La formule 5.3 exprime le fait que tous les opérandes de $expr$ ne sont pas représentés sur le chemin. Dans le cadre d'un DDD ordonné suivant toutes les variables de E, ce cas ne peut se produire. L'apparition de feuilles $\top$ est le résultat d'une erreur d'implantation des homomorphismes. La formule 5.4 signifie que l'homomorphisme se propage sur les arcs sortant d'un nœud étiqueté par une variable n'intervenant pas dans l'expression booléenne, tandis la sélection porte sur une nouvelle expression booléenne, tenant compte de la valeur de e si celle-ci intervient dans $expr$.* □

Exemple 5.1 (Sélection d'états satisfaisant une expression booléenne)

$$\begin{aligned}
& \langle\ \mathsf{SelectExpr}\ (a+b=d)\ \rangle (a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1 \\ + & b \xrightarrow{0} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1 \end{pmatrix} \\
= \ & a \xrightarrow{1} \langle\ \mathsf{SelectExpr}\ (1+b=d)\ \rangle \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1 \\ + & b \xrightarrow{0} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1 \end{pmatrix} \\
= \ & a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} \langle\ \mathsf{SelectExpr}\ (1+1=d)\ \rangle (c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1) \\ + & b \xrightarrow{0} \langle\ \mathsf{SelectExpr}\ (1+0=d)\ \rangle (c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1) \end{pmatrix} \\
= \ & a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} \langle\ \mathsf{SelectExpr}\ (2=d)\ \rangle (d \xrightarrow{2} e \xrightarrow{3} 1) \\ + & b \xrightarrow{0} c \xrightarrow{1} \langle\ \mathsf{SelectExpr}\ (1=d)\ \rangle (d \xrightarrow{2} e \xrightarrow{3} 1) \end{pmatrix} \\
= \ & a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} \langle\ \mathsf{SelectExpr}\ (2=2)\ \rangle (e \xrightarrow{3} 1) \\ + & b \xrightarrow{0} c \xrightarrow{1} d \xrightarrow{2} \langle\ \mathsf{SelectExpr}\ (1=2)\ \rangle (e \xrightarrow{3} 1) \end{pmatrix} \\
= \ & a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} \Big(\langle\ \mathsf{SelectExpr}\ (vrai)\ \rangle (e \xrightarrow{3} 1)\Big) \\ + & b \xrightarrow{0} c \xrightarrow{1} d \xrightarrow{2} \Big(\langle\ \mathsf{SelectExpr}\ (faux)\ \rangle (e \xrightarrow{3} 1)\Big) \end{pmatrix} \\
= \ & a \xrightarrow{1} \begin{pmatrix} & b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1 \\ + & b \xrightarrow{0} c \xrightarrow{1} d \xrightarrow{2} 0 \end{pmatrix} \\
= \ & a \xrightarrow{1} b \xrightarrow{1} c \xrightarrow{1} d \xrightarrow{2} e \xrightarrow{3} 1
\end{aligned}$$

5.4.2 Affectations

De tous les homomorphismes réalisés dans notre approche, celui qui réalise l'affectation d'une variable par une expression entière est le plus important à définir. En effet, il met en jeu tous les mécanismes nécessaires à l'élaboration des autres homomorphismes. A l'instar de $\langle\ \mathsf{selectExpr}\ ()\ \rangle$, son fonctionnement consiste à collecter les valeurs nécessaires à l'évaluation de l'expression à affecter, puis à affecter la valeur de l'expression à la variable d'affectation.

Proposition 5.2 (Homomorphisme $\langle\ \mathsf{setExpr}\ ()\ \rangle$) *L'homomorphisme $\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle$ affecte la valeur de l'expression $expr$ à la variable var. Il est défini par les règles suivantes :*

$$\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle (1) = \top \tag{5.5}$$

$$\langle\ \mathsf{setExpr}\ (var, expr = cst)\ \rangle = \langle\ \mathsf{setCst}\ (cst)\ \rangle \tag{5.6}$$

$$\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle (e \neq var, x) = \begin{cases} e \xrightarrow{x} \langle\ \mathsf{setExpr}\ (expr_{e:=x})\ \rangle & \text{si } e \in expr \\ e \xrightarrow{x} \langle\ \mathsf{setExpr}\ (expr)\ \rangle & \text{sinon} \end{cases} \tag{5.7}$$

$$\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle (e = var, x) = \begin{cases} \begin{pmatrix} \langle\ \mathsf{replace}\ (s)\ \rangle \\ \circ \langle\ \mathsf{collectExpr}\ (s, expr_{e:=x})\ \rangle \end{pmatrix} & \text{si } e \in expr \\ \\ \begin{pmatrix} \langle\ \mathsf{replace}\ (s)\ \rangle \\ \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle \end{pmatrix} & \text{sinon} \end{cases} \tag{5.8}$$

Explication *: L'équation 5.5 signifie que le résultat d'une affectation n'est indéfini que si l'on a rencontré var et tous les opérandes de $expr$. L'équation 5.6 transforme l'homomorphisme en affectation constante quand l'expression $expr$ peut être évaluée ($\langle$ setCst $()$ $\rangle$ est défini au chapitre 2). L'équation 5.7 donne le principe utilisé pour collecter toutes les informations nécessaires à l'évaluation de $expr$ dans le but de passer à une affectation constante.*

Malheureusement, cette collecte peut ne pas être achevée lorsque la variable à modifier est atteinte. On continue alors la collecte (équation 5.8) dans le but de stocker la valeur affectée une fois la collecte achevée, puis de repositionner la variable dans le DAG, à l'aide de l'homomorphisme $\langle$ replace $()$ $\rangle$ défini au chapitre 4. Le mécanisme de collecte intervenant après que la variable à modifier ait été rencontrée, est défini comme suit :

$$\begin{aligned}
\langle \text{ collectExpr } (var, expr \neq cst) \rangle(1) &= \top \\
\langle \text{ collectExpr } (var, expr = cst) \rangle(e,x) &= var \xrightarrow{cst} e \xrightarrow{x} \langle \text{ id } \rangle \\
\langle \text{ collectExpr } (var, expr \neq cst) \rangle(e,x) &= \begin{cases} e \xrightarrow{x} \langle \text{ collectExpr } (s, expr_{e:=x}) \rangle & \text{si } e \in expr \\ e \xrightarrow{x} \langle \text{ collectExpr } (s, expr) \rangle & \text{sinon} \end{cases}
\end{aligned}$$

□

La composition des homomorphismes $\langle$ replace $()$ $\rangle$ et $\langle$ collectExpr $()$ $\rangle$ est réalisée à l'aide de deux parcours du DAG. L'homomorphisme $\langle$ downExpr $()$ $\rangle$ réalise le même traitement en un seul parcours du DDD :

Proposition 5.3 (Homomorphisme $\langle$ downExpr $()$ $\rangle$) *L'homomorphisme $\langle$ downExpr $(var, expr)$ $\rangle$ défini par*

$$\begin{aligned}
\langle \text{ downExpr } (s, expr) \rangle(1) &= \top \\
\langle \text{ downExpr } (s, expr = cst) \rangle(e,x) &= var \xrightarrow{cst} e \xrightarrow{x} \langle \text{ id } \rangle \\
\langle \text{ downExpr } (s, expr \neq cst) \rangle(e,x) &= \begin{cases} \langle \text{ up } (e,x) \rangle \circ \langle \text{ downExpr } (s, expr_{e:=x}) \rangle & \text{si } e \in expr \\ \langle \text{ up } (e,x) \rangle \circ \langle \text{ downExpr } (s, expr) \rangle & \text{sinon} \end{cases}
\end{aligned}$$

réalise l'homomorphisme $\langle$ replace (s) $\rangle \circ \langle$ collectExpr $(s, expr)$ $\rangle$ et préserve donc l'ordre initial du DDD.

Démonstration *: En annexe page 128.* □

Pour résumer, l'évaluation de l'homomorphisme $\langle$ setExpr $(var, expr)$ $\rangle$ sur la feuille 1 est indéfinie ($\top$), et pour tout couple (e,x), elle est donnée dans le tableau 5.2 :

Exemple 5.2 (utilisation de $\langle$ up $()$ $\rangle$ et $\langle$ downExpr $()$ $\rangle$) *On cherche à effectuer l'affectation $b = a + b + d$ sur le DDD $a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1$.*

$$\begin{aligned}
& \langle \text{ setExpr } (b, a+b+d) \rangle(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{ setExpr } (b, 1+b+d) \rangle(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{ downExpr } (b, 1+2+d) \rangle(c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{ up } (c,3) \rangle \circ \langle \text{ downExpr } (b, 3+d) \rangle(d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{ up } (c,3) \rangle(b \xrightarrow{7} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} b \xrightarrow{7} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1
\end{aligned}$$

$\langle$ setExpr $(var, expr)$ $\rangle(e, x)$	$e \neq var$	$e = var$
$e \notin expr$	$e \xrightarrow{x} \langle$ this $\rangle$	$\langle$ downExpr $(var, expr)$ $\rangle$
$e \in expr$	$e \xrightarrow{x} \langle$ setExpr $(var, expr_{e:=x})$ $\rangle$	$\langle$ downExpr $(var, expr_{e:=x})$ $\rangle$

TAB. 5.2 Evaluation inductive de $\langle$ setExpr () $\rangle$

5.4.3 Ecritures-Lectures dans les canaux de communication

Introduction

Les canaux de communication en Promela sont de taille fixe. Ici on suppose qu'il ne peut transiter que des entiers. Nous avons choisi de représenter les canaux comme un conteneur dont le premier élément serait la taille du canal et les éléments suivants classés par ordre d'insertion dans le canal. Les opérations sur les canaux en Promela sont gardées, de ce fait, il est impossible de lire (resp. écrire) une donnée dans un canal vide (resp. plein). Lorsque l'instruction est exécutée, une valeur est soit retirée soit insérée dans le canal, les autres valeurs présentes dans le conteneur ne sont pas modifiées.

Tests sur l'état du canal

L'homomorphisme $\langle$ notEmpty (q) $\rangle$ sélectionne les états pour lesquels le canal q n'est pas vide (il suffit de vérifier que le la deuxième occurence de q ne vaut pas $\sharp$) :

$$\begin{aligned}
\langle \text{ notEmpty } (q) \rangle(1) &= \top \\
\langle \text{ notEmpty } (q) \rangle(e \neq q, x) &= e \xrightarrow{x} \langle \text{ this } \rangle \\
\langle \text{ notEmpty } (q) \rangle(q, x) &= q \xrightarrow{x} \langle \text{ selectExpr } (q \neq \sharp) \rangle
\end{aligned}$$

L'homomorphisme $\langle$ notFull (q) $\rangle$ sélectionne les états pour lesquels le canal q n'est pas plein (il utilise l'homomorphisme $\langle$ remainPlace (n) $\rangle$ qui vérifie qu'il reste au moins une place parmi les n disponibles dans q) :

$$\begin{aligned}
\langle \text{ notFull } (q) \rangle(1) &= \top \\
\langle \text{ notFull } (q) \rangle(e \neq q, x) &= e \xrightarrow{x} \langle \text{ this } \rangle \\
\langle \text{ notFull } (q) \rangle(q, size) &= q \xrightarrow{size} \langle \text{ remainPlace } (size) \rangle \\
\langle \text{ remainPlace } (n) \rangle(q, x) &= \begin{cases} 0 & \text{si } n = 0 \\ \langle \text{ id } \rangle & \text{si } (n \neq 0) \wedge (x = \sharp) \\ q \xrightarrow{x} \langle \text{ remainPlace } (n-1) \rangle & \text{sinon} \end{cases}
\end{aligned}$$

Ecriture d'une expression dans un canal

Le mécanisme d'écriture est très proche de celui de l'affectation d'une variable scalaire. La différence tient au seul fait que la valeur de l'expression est *insérée* : elle ne modifie l'étiquette d'aucun arc sortant préexistant. La valeur de l'expression est insérée avant le délimiteur de fin de file ($\sharp$).

Comme pour l'affectation d'expression on construit d'abord l'homomorphisme d'écriture de valeur constantes :

$$\begin{array}{rcl}
\langle\ \mathsf{writeCst}\ (q, cst)\ \rangle(1) & = & \top \\
\langle\ \mathsf{writeCst}\ (q, cst)\ \rangle(e \neq q, x) & = & e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle \\
\langle\ \mathsf{writeCst}\ (q, cst)\ \rangle(q, x) & = & q \xrightarrow{x} \langle\ \mathsf{insertCst}\ (cst)\ \rangle \\
\langle\ \mathsf{insertCst}\ (cst)\ \rangle(q, x) & = & \left\{ \begin{array}{rl} q \xrightarrow{x} \langle\ \mathsf{this}\ \rangle & \text{si } x \neq \sharp \\ q \xrightarrow{cst} q \xrightarrow{\sharp} \langle\ \mathsf{id}\ \rangle & \text{sinon} \end{array} \right.
\end{array}$$

On procède au même principe de collecte que pour l'affectation :

$$\begin{array}{rcl}
\langle\ \mathsf{writeExpr}\ (q, expr)\ \rangle(1) & = & \top \\
\langle\ \mathsf{writeExpr}\ (q, expr = cst)\ \rangle & = & \langle\ \mathsf{writeCst}\ (q, cst)\ \rangle \\
\langle\ \mathsf{writeExpr}\ (q, expr)\ \rangle(e \neq q, x) & = & \left\{ \begin{array}{rl} e \xrightarrow{x} \langle\ \mathsf{writeExpr}\ (q, expr_{e:=x})\ \rangle & \text{si } e \in expr \\ e \xrightarrow{x} \langle\ \mathsf{writeExpr}\ (q, expr)\ \rangle & \text{sinon} \end{array} \right.
\end{array}$$

Si l'expression $expr$ n'est pas complétée lorsque q est rencontrée, on s'appuie sur l'homomorphisme $\langle\ \mathsf{downExpr}\ ()\ \rangle$:

$$\begin{array}{rcl}
\langle\ \mathsf{writeExpr}\ (q, exp)\ \rangle(q, x) & = & q \xrightarrow{x} \langle\ \mathsf{insertExpr}\ (expr)\ \rangle \\
\langle\ \mathsf{insertExpr}\ (expr)\ \rangle(q, x) & = & \left\{ \begin{array}{rl} e \xrightarrow{q} \langle\ \mathsf{this}\ \rangle & \text{si } x \neq \sharp \\ \langle\ \mathsf{up}\ (q, \sharp)\ \rangle \circ \langle\ \mathsf{downExpr(q, expr)}\ \rangle & \text{sinon} \end{array} \right.
\end{array}$$

Lecture d'une valeur dans un canal

La lecture dans un canal est beaucoup plus simple, il suffit de retirer le deuxième élément du conteneur (c'est-à-dire celui qui suit immédiatement la taille du canal), l'homomorphisme $\langle\ \mathsf{notEmpty}\ ()\ \rangle$ ayant préalablement assuré que la file est non vide. L'homomorphisme $\langle\ \mathsf{readVar}\ (q, var)\ \rangle$ est construit comme suit :

$$\begin{array}{rcl}
\langle\ \mathsf{readVar}\ (q, var)\ \rangle(1) & = & \top \\
\langle\ \mathsf{readVar}\ (q, var)\ \rangle(e, x) & = & \left\{ \begin{array}{rl} e \xrightarrow{x} \langle\ \mathsf{retract}\ (var)\ \rangle & \text{si } e = q \\ \langle\ \mathsf{downFifo}\ (q, var)\ \rangle & \text{si } e = var \\ e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle & \text{sinon} \end{array} \right. \\
 & \text{avec} & \\
\langle\ \mathsf{downFifo}\ (q, var)\ \rangle(e, x) & = & \left\{ \begin{array}{rl} \langle\ \mathsf{up}\ (e, x)\ \rangle \circ \langle\ \mathsf{this}\ \rangle & \text{si } e \neq q \\ \langle\ \mathsf{up}\ (e, x)\ \rangle \circ \langle\ \mathsf{retractUp}\ (var)\ \rangle & \text{sinon} \end{array} \right. \\
 & \text{et} & \\
\langle\ \mathsf{retractUp}\ (var)\ \rangle(q, x) & = & \left\{ \begin{array}{rl} var \xrightarrow{x} \langle\ \mathsf{id}\ \rangle & \text{s } x \neq \sharp \\ 0 & \text{sinon} \end{array} \right. \\
 & \text{et} & \\
\langle\ \mathsf{retract}\ (var)\ \rangle(q, x) & = & \left\{ \begin{array}{rl} \langle\ \mathsf{setCst}\ (var, x)\ \rangle & \text{si } x \neq \sharp \\ 0 & \text{sinon} \end{array} \right.
\end{array}$$

5.4.4 Opérations réciproques (pour l'opérateur pre)

Nous décrivons ici les homomorphismes spécifiques aux opérateurs pre du tableau 5.1. Il s'agit des homomorphismes $\langle\ \mathsf{preSetExpr}\ ()\ \rangle$, $\langle\ \mathsf{preWriteExpr}\ ()\ \rangle$ et $\langle\ \mathsf{preReadExpr}\ ()\ \rangle$ associés respectivement à l'affectation, l'écriture et la lecture dans un canal.

Ecriture dans un canal

L'opération réciproque de l'écriture de la valeur d'une expression dans un canal, qui peut se faire en retirant la dernière valeur insérée (c'est-à-dire celle qui précède l'apparition de la valeur $\sharp$) puis en sélectionnant les états pour lesquels cette valeur est égale à l'expression insérée. Le mécanisme est similaire à une selection d'états satisfaisant une expression booléenne et se particularise une fois le canal atteint :
Si $e \neq q$,

$$\begin{aligned}
\langle\ \mathsf{preWriteExpr}\ (q, expr)\ \rangle(e,x) &= \begin{cases} e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle & \text{si } e \notin expr \\ e \xrightarrow{x} \langle\ \mathsf{preWriteExpr}\ (q, expr_{e:=x})\ \rangle & \text{sinon} \end{cases} \\
\langle\ \mathsf{preWriteExpr}\ (q, expr)\ \rangle(q,x) &= e \xrightarrow{x} \langle\ \mathsf{retTailExpr}\ (expr)\ \rangle
\end{aligned}$$

L'homomorphisme $\langle\ \mathsf{retTailExpr}\ ()\ \rangle$ retire le dernier élément inséré dans le canal si ce dernier existe et est égal à $expr$.

$$\begin{aligned}
\langle\ \mathsf{retTailExpr}\ (expr)\ \rangle(q, x=\sharp) &= 0 \\
\langle\ \mathsf{retTailExpr}\ (expr)\ \rangle(q, x\neq\sharp) &= \left(\begin{array}{cc} & q \xrightarrow{x} \langle\ \mathsf{this}\ \rangle \\ + & \langle\ \mathsf{selectExpr}\ (expr = x \wedge (q=\sharp))\ \rangle \end{array}\right)
\end{aligned}$$

Cet homomorphisme propage deux chemins (d'où la somme $+$) à chaque nœud visité, car on ne peut pas savoir à l'avance si la prochaine affectation portera sur la valeur de fin de file $\sharp$.

Exemple 5.3 (Retrait du dernier élément d'un canal) *Dans cet exemple, on applique l'homomorphisme $\langle\ \mathsf{retTailExpr}\ (3)\ \rangle$ sur la file $q \xrightarrow{3} q \xrightarrow{2} q \xrightarrow{3} q \xrightarrow{\sharp} 1$:*

$$\begin{aligned}
& \langle\ \mathsf{retTailExpr}\ (3)\ \rangle(q \xrightarrow{3} q \xrightarrow{2} q \xrightarrow{3} q \xrightarrow{\sharp} 1) \\
= & \left(\begin{array}{cc} & q \xrightarrow{3} \langle\ \mathsf{retTailExpr}\ (3)\ \rangle(q \xrightarrow{2} q \xrightarrow{3} q \xrightarrow{\sharp} 1) \\ + & \underbrace{\Big(\langle\ \mathsf{selectExpr}\ ((3=3)\wedge(q=\sharp))\ \rangle(q \xrightarrow{2} q \xrightarrow{3} q \xrightarrow{\sharp} 1)\Big)}_{} = 0 \end{array}\right) \\
= & \ q \xrightarrow{3} \left(\begin{array}{cc} & q \xrightarrow{2} \langle\ \mathsf{retTailExpr}\ (3)\ \rangle(q \xrightarrow{3} q \xrightarrow{\sharp} 1) \\ + & \Big(\langle\ \mathsf{selectExpr}\ ((3=2)\wedge(q=\sharp))\ \rangle(q \xrightarrow{3} q \xrightarrow{\sharp} 1)\Big) = 0 \end{array}\right) \\
= & \ q \xrightarrow{3} q \xrightarrow{2} \left(\begin{array}{cc} & q \xrightarrow{3} \Big(\langle\ \mathsf{retTailExpr}\ (3)\ \rangle(q \xrightarrow{\sharp} 1)\Big) = 0 \\ + & \Big(\langle\ \mathsf{selectExpr}\ ((3=3)\wedge(q=\sharp))\ \rangle(q \xrightarrow{\sharp} 1)\Big) = q \xrightarrow{\sharp} 1 \end{array}\right) \\
= & \ q \xrightarrow{3} q \xrightarrow{2} q \xrightarrow{\sharp} 1
\end{aligned}$$

Affectation de constantes

L'ensemble des états prédécesseurs à l'exécution d'une instruction d'affectation $var = cst$ n'est pas déterminé si nous ne connaissons pas *a priori* la borne de var, ou mieux, l'ensemble des valeurs effectivement prises par var dans le contexte du franchissement de l'instruction $var = cst$.

Le calcul des prédécesseurs d'une affectation constante fait intervenir deux ensembles d'états :
- To, l'ensemble des états dont nous voulons connaître les prédécesseurs (par l'instruction $var = cst$)
- et $\mathcal{C}$, l'ensemble des états candidats qui est une approximation des états représentant le contexte d'exécution de $var = cst$. Il est contenu dans l'ensemble des états accessibles.

Nous partons des hypothèses suivantes :
- La valeur initiale de la variable modifiée peut être quelconque.
- La seule variable modifiée est var.

La solution peut se décomposer en trois traitements :
1. Sélectionner dans le DDD To les états pour lesquels $var = cst$.
2. Étendre dans ce nouveau DDD la valeur de var à toutes les valeurs possibles *proposées par les états candidats* (car on ne connait pas la borne de var).
3. Procéder à une intersection de ce dernier DDD avec l'ensemble des candidats.

L'homomorphisme $sweet$ réalise ces trois traitements en un seul parcours. Il est paramétré par var, cst et les états dont on veut calculer les prédécesseurs To. Il s'applique à l'ensemble des états candidats $\mathcal{C}$:

$$\begin{aligned} & \langle\ \mathsf{preSetCst}\ (var, cst, \mathcal{C}\ \rangle(To) \\ = & \langle\ \mathsf{sweet}\ (var, cst, To)\ \rangle(\mathcal{C}) \end{aligned}$$

On détaille ici l'application locale de l'homomorphisme $\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle$ à un nœud quelconque (e, x). Plusieurs cas se présentent :

1. Il n'existe pas de prédécesseur (ni de successeur) à un Tuple *absent* :

$$\langle\ \mathsf{sweet}\ (var, cst, To = 0)\ \rangle = 0$$

2. Tant qu'il n'a pas rencontré var, il procède à une intersection locale :

$$\begin{aligned} & \langle\ \mathsf{sweet}\ (var, cst, To)\ \rangle(e \neq var, x) \\ = \ & e \xrightarrow{x} \langle\ \mathsf{sweet}\ (var, cst, To|_{e=x})\ \rangle \end{aligned}$$

   où $To|_{e=x}$ désigne le nœud pointé par l'arc issu de la racine de e, étiqueté par x.

3. Une fois var rencontrée, on étend son ensemble de valeurs prises à toutes celles qui figurent dans l'ensemble des états candidats (plus précisément toutes les valeurs étiquetant les arcs issus du nœud courant). A partir de là, l'intersection (pour les variables restantes) se fait par un produit classique de DDD.

$$\begin{aligned} & \langle\ \mathsf{sweet}\ (var, cst, To)\ \rangle(e, x) \\ = \ & e \xrightarrow{x} (\langle\ \mathsf{id}\ \rangle \times To|_{e=cst}) \end{aligned}$$

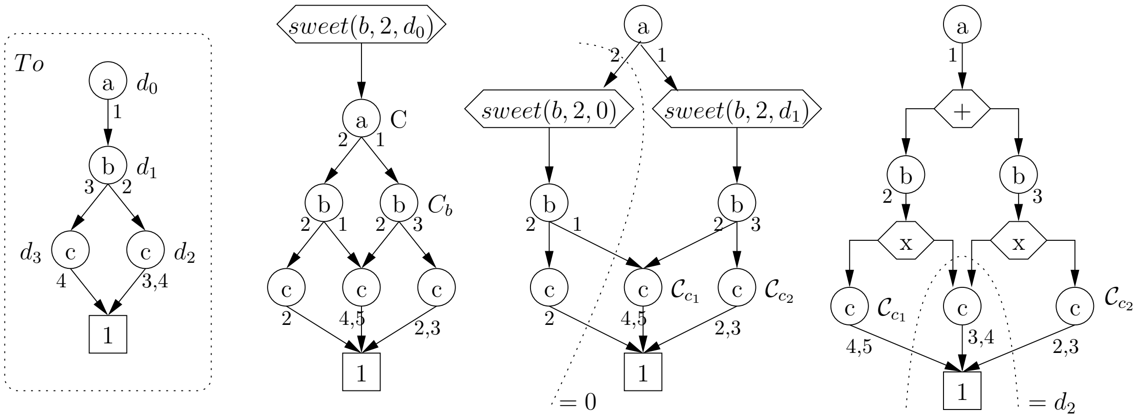


FIG. 5.1 Évaluation de l'opérateur $sweet$

Exemple 5.4 *Une évaluation de l'opérateur $sweet$ est donnée sur la figure 5.1. Dans cet exemple, on cherche à calculer l'ensemble des états prédécesseurs de To par l'application de $b = 2$, sur l'ensemble des candidats désignés par $\mathcal{C}$. On applique à $\mathcal{C}$ l'homomorphisme $\langle$ sweet $(b, 2, To)$ $\rangle$. Ce qui revient à effectuer un parcours attelé entre To et $\mathcal{C}$ jusqu'à rencontrer la variable b. Le DDD To n'accepte pas $a = 2$, donc l'application de sweet sur $a \xrightarrow{2}$ ... s'évalue en une feuille nulle, et l'homomorphisme se propage dans $\mathcal{C}$ sur l'arc $a \xrightarrow{1} C_b$. Par définition, toutes les valeurs des arcs sortants du nœud C_b sont acceptables. L'ensemble des états dont elles sont le support $((b \xrightarrow{2} C_{c_1}) + (b \xrightarrow{3} C_{c_2}))$ est intersecté avec les états de To représentant le contexte $b = 2$ ($b \xrightarrow{2} d_2$) à partir des nœuds fils (quatrième schéma).*

Le résultat est donné à droite. On a bien sélectionné dans C les états prédécesseurs de To par le franchissement de $b = 2$ respectant le contexte d'exécution de cette instruction.

Affectations d'expressions

L'opérateur pre associé à l'affectation d'une expression ($\langle$ preSetExpr () $\rangle$) est construit à l'aide de $\langle$ preSetCst () $\rangle$. On garde toujours le principe d'un parcours attelé pour étendre les valeurs de la variable affectées à celles du contexte d'exécution donné par l'ensemble des états candidats. Dans le cas d'une affectation de constante on s'assurait que la variable affectée avait bien la valeur constante définie. Ici on vérifie que la valeur de la variable affectée est égale à celle de l'expression que l'on affecte. On construit l'homomorphisme $\langle$ preSetExpr $(var, expr, \mathcal{C})$ $\rangle$ comme suit :

$$\begin{aligned}
\langle \text{ preSetExpr } (var, expr, \mathcal{C}) \rangle(1) &= \top \\
\langle \text{ preSetExpr } (var, expr, 0) \rangle(d) &= 0 \\
\langle \text{ preSetExpr } (var, expr = cst, \mathcal{C}) \rangle(d) &= \langle \text{ preSetCst } (var, cst, \mathcal{C}) \rangle(d)
\end{aligned}$$

Et si $e \neq var$ et $expr \neq cst$,

$$\langle \text{ preSetExpr } (var, expr, \mathcal{C}) \rangle(e, x) = \begin{cases} e \xrightarrow{x} \langle \text{ preSetExpr } (var, expr_{e:=x}, \mathcal{C}|_{e=x}) \rangle & \text{si } e \in expr \\ e \xrightarrow{x} \langle \text{ preSetExpr } (var, expr, \mathcal{C}|_{e=x}) \rangle & \text{sinon} \end{cases}$$

Dans le cas où l'expression n'est pas complétée lorsque l'homomorphisme rencontre la variable affectée, on réutilise l'homomorphisme $\langle\ \mathsf{preSetCst}\ ()\ \rangle$ et pour chaque valeur x de var on sélectionne les sous-ensembles pour lesquels $x = expr$:

$$\langle\ \mathsf{preSetExpr}\ (var, expr, \mathcal{C})\ \rangle(var, x) \quad = \quad \begin{pmatrix} \langle\ \mathsf{selectExpr}\ (expr = x)\ \rangle \\ \circ\langle\ \mathsf{preSetCst}\ (var, x, \mathcal{C})\ \rangle \\ \circ(var \xrightarrow{x} \langle\ \mathsf{id}\ \rangle) \end{pmatrix}$$

On note que cette fois, le traitement ne se fait pas en un seul parcours puisqu'il compose deux homomorphismes censés se propager ($\langle\ \mathsf{selecExpr}\ ()\ \rangle$ et $\langle\ \mathsf{preSetCst}\ ()\ \rangle$). Il est *possible* néanmoins de remplacer cette composition en un unique parcours, mais celà suppose la définition d'un homomorphisme basé sur $\langle\ \mathsf{sweet}\ ()\ \rangle$ et particularisé avec une sélection de d'états satisfaisant une expression booléenne. Or nous verrons en section 5.5.3 que l'évaluation de l'opérateur pre n'est pas un facteur de complexité en comparaison avec l'opérateur $post$. Aussi, nous n'avons pas cherché à optimiser ce traitement.

Lectures dans un canal

La lecture d'une variable var dans un canal q consiste à retirer le premier élément de la file et d'en affecter sa valeur à var. L'opération réciproque se décompose en trois traitements :

- S'assurer que le canal n'est pas plein. Les états ne satisfaisant pas cette condition ne peuvent pas être consécutifs à une lecture.
- Etendre var à l'ensemble des valeurs respectant le contexte d'exécution de l'instruction ; ce qui s'apparente à une affectation de constante.
- Insérer la valeur prise par var à la tête de q.

On se dote d'abord d'un homomorphisme capable d'ajouter une valeur constante (cst) en tête de canal (on suppose que celui-ci n'est pas plein) :

$$\langle\ \mathsf{addHeadCst}\ (q, cst)\ \rangle(e, x) \quad = \quad \begin{cases} e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle & \text{si } e \neq q \\ e \xrightarrow{x} q \xrightarrow{cst} \langle\ \mathsf{id}\ \rangle & \text{sinon} \end{cases}$$

L'homomorphisme $\langle\ \mathsf{preReadVar}\ ()\ \rangle$ est alors défini comme suit :

$$\begin{aligned} & \langle\ \mathsf{preReadVar}\ (var, q, \mathcal{C})\ \rangle(e, x) \\ = & \begin{cases} e \xrightarrow{x} \langle\ \mathsf{preSetCst}\ (var, q, \mathcal{C}|_{e=x})\ \rangle \circ \langle\ \mathsf{downExpr}\ (f, v)\ \rangle & \text{si } e = q \\ \langle\ \mathsf{preSetCst}\ (var, x, \mathcal{C})\ \rangle \circ e \xrightarrow{x} \langle\ \mathsf{addHeadCst}\ (q, x)\ \rangle & \text{si } e = var \\ e \xrightarrow{x} \langle\ \mathsf{preReadVar}\ (var, q, \mathcal{C}|_{e=x})\ \rangle & \text{sinon} \end{cases} \end{aligned}$$

On note ici que le traitement ne peut pas se traduire par un parcours attelé unique. En effet, l'évaluation de l'homomorphisme $\langle\ \mathsf{preSetCst}\ ()\ \rangle$ doit être paramétrée par un contexte où la valeur de var *a été insérée* en tête de file. Il nous faut donc évaluer ce contexte pour pouvoir étendre le domaine de var à l'ensemble des valeurs qui lui correspondent.

5.5 Premières conclusions

Dans cette section nous étudions les performances d'un prototype de model-checker symbolique, utilisant les DDD et les homomorphismes décrits dans ce chapitre, pour les systèmes respectant le sous ensemble de Promela défini au chapitre 3. La comparaison avec les performances de SPIN permet de dégager les classes de systèmes pour lesquelles les méthodes symboliques sont les plus efficaces. Afin de statuer sur l'efficacité des DDD, par rapport aux BDD, nous avons implanté les opérateurs $select$, $post$ et pre sur une bibliothèque BDD et comparé les performances des deux approches.

5.5.1 Comparaison avec SPIN

La comparaison porte sur la construction de l'ensemble des états accessibles.

- SPIN vérifie la propriété $\mathbf{G}vrai$ sur le programme Promela en employant les options que SPIN suggère pour optimiser le calcul.
- Notre prototype construit l'ensemble des états accessibles, avec l'algorithme 2.1, avant même de procéder aux vérifications de propriétés CTL.

Remarques

1. La propriété $\mathbf{G}vrai$ produit un automate de Büchi à un seul état. La synchronisation de ce dernier avec l'automate du système ne fausse pas la taille de l'ensemble des états à explorer. De plus cette propriété étant satisfaite sur tous les états, SPIN ne peut trouver de contre-exemple et interrompre prématurément le calcul de l'ensemble d'accessibilité.
2. La vérification de propriétés communes aux logiques CTL et LTL auraient donné des résultats difficiles à interpréter :
   - la taille de l'automate de Büchi croît à mesure que la propriété à vérifier devient complexe. De tels cas de figure désavantageraient considérablement SPIN.
   - Si la propriété à vérifier est fausse, SPIN conclut au premier contre-exemple trouvé ce qui lui donne un avantage « artificiel » par rapport à notre outil qui détaille l'ensemble des états satisfaisant la propriété.

5.5.2 Représentation sur BDD

Afin de statuer sur l'apport des DDD et des méthodes inductives par rapport aux méthodes symboliques existantes, nous avons porté les méthodes décrites dans ce chapitre sur une bibliothèque BDD. Le fonctionnement global de l'outil demeurant le même, la comparaison des deux approches permet d'évaluer les performances intrinsèques des deux représentations pour un système donné.

La bibliothèque BDD

La manipulation des BDD s'est faite en utilisant la bibliothèque Buddy [Bud] développée à l'université de Copenhague [LNA99]. La construction des espaces d'états et des opérateurs $post$ et pre sur BDD a été réalisée par Sami Taktak [Tak04].

La bibliothèque Buddy fournit un ensemble riche de fonctions sur BDD, notamment des stratégies de *réordonnancement dynamiques*. Enfin, la manipulation de valeurs entières est sensiblement facilitée par l'emploi de vecteurs de BDD pour la représentation de domaines finis.

Problèmes de représentation en BDD des configurations de variables avec Buddy

La construction bit-à-bit des variables représentées sur un BDD pose le problème du nombre de bits nécessaire à l'encodage de chaque variable Promela. En effet, plus il y aura de bits utilisés pour encoder une variable, plus la profondeur du BDD sera importante, ce qui se traduit par une dégradation des performances (temps et occupation mémoire). Nous avons donc employé des modèles où les variables étaient encodées suivant un nombre de bits choisi par l'utilisateur (`unsigned`) aussi souvent que possible.

L'encodage bit-à-bit des variables pose un problème de compatibilité de domaines de définition auquel les DDD sont insensibles : La manipulation des valeurs et expressions entières a beau être facilitée par l'emploi de vecteurs de BDD, Buddy interdit les opérations binaires entre deux valeurs encodées sur un nombre de bits différents. Sur les DDD, les expressions sur tableaux ont été évaluées avec un produit orthogonal ($t[n] = \sum_{i=0}^{n-1} t[i] \times (i = n)$) ce qui est impossible avec les BDD qui nécessitent des opérandes de même domaine pour construire ses opérateurs, aussi nous avons dû modifier les instructions Promela pour simuler, sans modifier le comportement global du système, ce produit orthogonal afin de le rendre compatible avec l'interface de Buddy.

La représentation des canaux est de loin le point critique de l'approche Promela-BDD : La lecture et l'écriture dans un canal, contrairement aux autres instructions élémentaires, ne se réduisent pas à l'affectation d'une variable ou une sélection d'états. Les canaux étant de taille limitée en Promela, ils ont été représentés par des tableaux dont il a fallu définir une politique d'accès :

- Soit l'accès au tableau se fait de façon circulaire, ce qui est simple à mettre en œuvre mais multiplie par la taille du canal le nombre de représentations équivalentes pour un même état (problème de canonicité).
- Soit les valeurs sont rangées dans l'ordre d'insertion et décalées après une lecture afin que la plus ancienne soit située dans la première case du tableau. Cette méthode garantit une représentation canonique mais nécessite un mécanisme de recopie (pour le décalage) qui peut être extrêmement lourd.

La librairie [Bud] offre des modules de remplacement de variables efficaces qui ont permis de mettre en œuvre la seconde méthode. Ainsi, nous avons pu générer exactement les mêmes espaces d'états sur BDD et sur DDD pour comparer ces deux structures.

5.5.3 Résultats

Ici, nous comparons les performances de l'outil SPIN avec celles des méthodes symboliques utilisant les DDD ou les BDD. Les résultats comparatifs sont donnés sur le tableau 5.3. Ils ont été obtenus sur un Processeur Intel Pentium IV cadencé à 3,2GHz, les calculs durant plus d'une journée ou exigeant plus qu'un Go de mémoire ayant été interrompus automatiquement. Les durées sont données en secondes (0 lorsque le le temps écoulé est inférieur à une seconde) et la mémoire utilisée est exprimée en mégaoctets. La colonne SPIN donne les performances de l'outil SPIN pour la construction de l'ensemble des états accessibles, les colonnes BDD, DDD et DDD-O donnent les performances obtenues avec des méthodes symboliques utilisant respectivement les BDD, les DDD munis d'un ordre quelconque (issu de l'ordre de déclarations dans le fichier Promela) et les DDD munis d'un ordre tel que la topologie du système le suggérait. Pour chaque outil, la colonne *reach* donne le temps nécessaire à la génération de l'espace des états accessibles, la colonne *check* donne le temps nécessaire à la vérification des formules CTL indiquées, et la colonne *mem* indique la mémoire nécessaire maximale pour mener ces calculs à terme. Pour être plus précis, quel que soit le système, et quelle que soit la propriété vérifiée, le « pic » d'utilisation mémoire a été atteint uniquement lors du calcul d'accessibilité (une fois le pic atteint lors du calcul d'accessibilité, la mémoire nécessaire aux calculs de vérification est beaucoup plus faible).

Les systèmes étudiés se regroupent en trois catégories :

- Les systèmes à très forte concurrence sans expressions arithmétiques comme le dîner des philosophes (*Dining Philosophers*).
- Les systèmes très faiblement concurrents comportant des expressions arithmétiques simples comme le protocole *Sliding Window* et dans une moindre mesure l'élection sur un anneau (*Leader Election*).
- Les systèmes comportant des expressions arithmétiques complexes, qui font intervenir pratiquement toutes les variables partagées du système, sans possibilité trouver un bon ordre naturel des variables (algorithmes de *Peterson* et du Boulanger (*Bakery*)).

Tous les programmes Promela sont donnés dans l'annexe D. Les résultats portant sur les DDD ont été obtenus sans aucune optimisation particulière (comme le réordonnancement dynamique, la réduction d'ordre partiel, la saturation de processus etc.) mis à part l'utilisation du cache de calcul. En revanche, pour les résultats portants sur les BDD nous avons employé toutes les optimisations de Buddy (les algorithmes de réordonnancement *Sift* [R. 93] et *Win2ite* [BRB90] en particulier), et ceux de SPIN ont été obtenus en tenant compte des paramètres suggérés *a posteriori* par l'outil (ce qui nous a mené à reconstruire entièrement l'espace d'états pour avoir les meilleures performances).

5.5.4 Discussion

Méthodes symboliques et énumératives

Dans cette partie, nous ne comparons que les performances concernant le calcul de l'ensemble des états accessibles. Pour les systèmes de petite taille (de l'ordre du million d'états), et très peu concurrents (comme le protocole *Sliding Window*), l'outil SPIN présente les meilleures performances à la fois en terme de mémoire et de temps CPU. En revanche, plus la taille du système est importante, plus les BDD et les DDD sont efficaces pour contenir l'explosion combinatoire.

Cette propriété est mise en évidence par la comparaison de trois systèmes différents produisant le même nombre total d'états :

- Le dîner des philosophes où les processus (ici) sont de très faible complexité (4 états par processus),
- Le protocole *Sliding Window* ou les processus sont peu nombreux mais beaucoup plus complexes (plusieurs centaines d'états par processus),
- et l'élection sur un anneau, dont la complexité des processus est intermédiaire.

La figure 5.2 donne (sur une échelle logarithmique) le nombre de *nouveaux* états générés, à chaque application de l'opérateur $post$. Le protocole *Sliding Window* ne « produit » pas plus de $250,000$ états par itération ce qui rend contre-productives les méthodes symboliques. D'un autre côté, pour les systèmes qui mènent à une génération d'un nombre important d'états par itération (comme l'élection ou le dîner qui peut aller jusqu'à 3.10^6 nouveaux états), les méthodes symboliques s'avèrent plus efficaces, même si des stratégies d'ordonnancement doivent être utilisées pour gagner un ordre de grandeur sur les méthodes explicites de représentation.

		SPIN		BDD			DDD			DDD-O		
N	$\vert S\vert$	reach	mem	reach	check	mem.	reach	check	mem	reach	check	mem
Dining Philosophers $\mathbf{AG}(\mathbf{AF}(eating\ philosopher))$												
5	244	0	2.3	0	0	93	0	0	4.3	0	0	5.5
10	5.9e4	2.3	36.7	4	6	94	31	170	17.6	9	2	11
11	1.7e5	7.6	41.6	9	15	94	82	935	30	14	2	12.8
12	5.3e5	29.1	90.1	23	42	94	277	6652	57	21	3	16.6
13	1.6e6	193	114.8	55	117	94	1021	3.7e4	105	30	3	19.6
14	4.8e6	660	737	137	337	95	4534	$> 24h$	203	41	4	24.4
15	1.4e7	2251	1067	380	1231	96	18375	$> 24h$	397	54	5	29.6
20	3.5e9	*	>1GB	*	*	*	*	*	>1GB	182	9	64.1
30	2.1e14	*	>1GB	*	*	*	*	*	>1GB	962	22	214
50	7.2e23	*	>1GB	*	*	*	*	*	>1GB	8006	67	994
Leader Election $\mathbf{AF}(best\ candidate\ elected)$												
5	5.4e3	10	23	510	191	94	8	1	6.3	11	1	6.2
6	3.2e4	262	144	5056	5394	144	34	5	11	37	3	11
7	1.8e5	*	>1GB	*	*	*	127	22	28	95	13	19
10	3.3e7	*	>1GB	*	*	*	4533	*	663	716	*	77
Sliding Window $\mathbf{AG}(\mathbf{AF}(new\ message\ sent))$												
2	4.0e5	0	8.1	552	221	102	486	142	17.8	662	206	17
3	3.5e7	29	450	65429	63216	332	4.8e5	$> 24h$	565	4.8e5	$> 24h$	475
4	*	*	>1GB	*	*	*	*	*	>1GB	*	*	>1GB
Peterson $\mathbf{AF}(query_i\ satisfied)$ and $\mathbf{AG}(query_i \Rightarrow \mathbf{AF}(critical_i))$												
2	208	0	5.5	0	1	94	0	0	3.6			
3	2.5e4	0	5.8	17	39	93	7	4	5.9			
4	6.4e6	20	402	1674	1093	400	1278	216	113			
5	*	*	>1GB	4.7e5	>24h	900	*	*	>1GB			
Bakery $\mathbf{AF}(client_i\ satisfied)$												
4	1.8e5	0	5.2	50	0	94	17	0	8.9			
5	1.6e7	1385	6.4	1231	0	93	379	5	46			
6	2.2e9	>24h	*	*	*	*	61901	47	455			

TAB. 5.3 Statistiques sur SPIN, DDDs et BDDs

La bonne performance des méthodes explicites ne s'explique pas seulement par le faible degré de concurrence des systèmes, la complexité des types de données est également un facteur important : le prototype utilisé pour les BDD et les DDD met à plat tous les types structurés, ce qui multiplie le nombre de variables sur les chemins du DAG ainsi que le nombre de transitions à effectuer notamment lorsque la valeur d'une variable structurée est insérée ou lue dans un canal. L'accroissement de la profondeur du DAG multiplie le nombre de nœuds dans le cache de calcul ce qui ralentit son accès (malgré l'utilisation d'une table de hachage) et accroit les ressources mémoire nécessaires. En effet, nous avons vu dans la section 2.3.3 que l'évaluation paresseuse des opérateurs sur les DDD s'effectuait par la conservation, dans le cache de calcul, de nœuds représentant les calculs réalisés. La taille de ce cache est donc d'autant plus grande qu'il y a de nœuds rencontrés sur un chemin. Un emploi plus fréquent du ramasse-miettes ou la supression du cache de calcul peut résoudre ce problème d'accroissement de la mémoire utilisée au détriment d'un temps de calcul plus important.

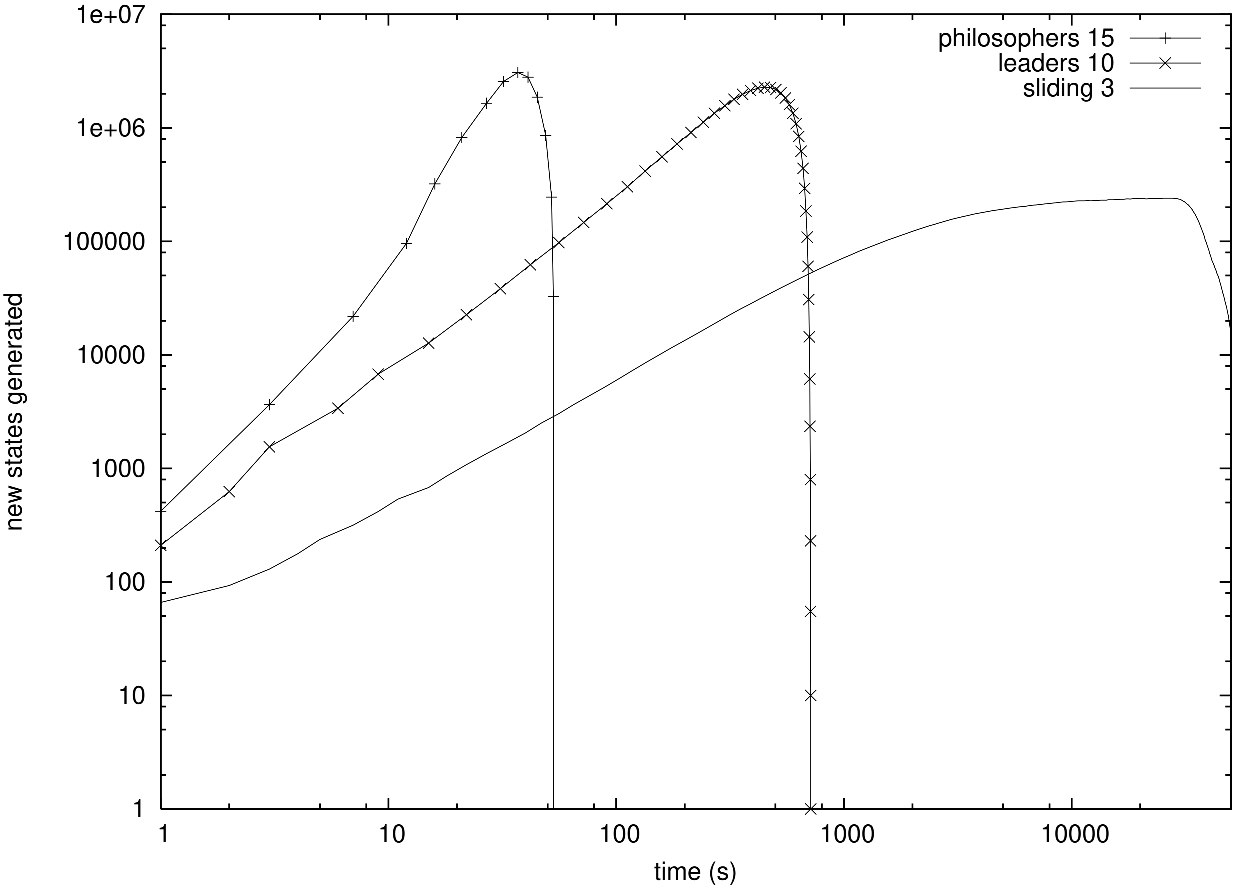


FIG. 5.2 Nouveaux états générés par application de l'opérateur $post$

BDD et DDD

Dans les exemples utilisés, tous les domaines des variables peuvent être connus *a priori*, et sont encodables sur les BDD ou les DDD. Les DDD présentent une bonne flexibilité sur l'emploi des fonctions arithmétiques utilisant des tableaux (Algorithmes de Peterson et du Boulanger) mais l'emploi d'expressions utilisant beaucoup d'opérandes rend les BDD légèrement plus efficaces que les DDD (un pas de plus pour l'algorithme de Peterson).

Sans ordre particulier sur les variables, les DDD ont des performances comparables avec les BDD utilisant la meilleure configuration d'ordonnancement, et lorsque l'on force un ordre "naturel", les DDD supplantent les BDD. On note que l'ordre efficace pour les DDD ne l'est pas pour les BDD.

Concernant les formules CTL, et donc l'application de l'opérateur pre, les différences structurelles inclinent en faveur des DDD. Ce résultat montre que même si l'abstraction d'une variable est un procédé très simple (et peu coûteux) sur les BDD, le principe d'utilisation d'un *unique* parcours attelé évite les calculs de sélection et d'intersection qui doivent s'exécuter sur un BDD entier. Ce gain en faveur des DDD s'explique donc par l'utilisation de méthodes inductives et non plus seulement par les différences structurelles entre BDD et DDD.

5.6 Conclusion

Nous avons montré comment réaliser, à l'aide d'homomorphismes inductifs, les traitements élémentaires nécessaires pour la vérification automatique de propriétés CTL. Ceci a donné lieu à l'implémentation d'un premier prototype de model-checker dont nous avons évalué les performances.

La comparaison avec le model-checker SPIN nous a permis d'identifier les catégories de systèmes pour lesquelles les méthodes symboliques s'avèrent les plus efficaces. Nous avons montré que, pour une majorité de systèmes, les DDD sans optimisation *ad hoc* pouvaient supplanter les BDD même optimisés. Conformément à nos attentes, l'ordre d'apparition des variables sur un chemin est déterminant pour tirer parti des propriétés de partage des DDD. Nous avons fait ressortir qu'un « bon ordre » peut être directement inspiré de la topologie du système.

Néanmoins, l'efficacité des DDD pâtit de la longueur des chemins le long du DAG : l'instanciation de conteneurs et de variables vectorielles multiplie les nœuds sur le chemin ce qui en accroît la longueur. Les chapitres suivants proposent l'utilisation de DDD « hiérarchiques » pour résoudre ces problèmes.

Chapitre 6

DDD Hiérarchiques

Nous avons montré au chapitre précédent que l'emploi des DDD permettait de vérifier efficacement des systèmes écrits dans le sous-ensemble statique de Promela. Nous avons aussi fait apparaître que la longueur des chemins dans le DAG nuisait à l'efficacité des méthodes inductives. La solution consistant à représenter une variable vectorielle ou un conteneur à l'aide d'un nœud unique réduirait cette influence mais abolirait les propriétés de partage des DDD que nous souhaitons conserver.
Les Diagrammes de Décision d'Ensembles (SDD, [TM04, CTM05]) sont une alternative intéressante pour réduire la longueur des chemins : ils sont basés sur la « hiérarchisation » des données qui étend le partage aux étiquettes des arcs.
Le but de ce chapitre est de montrer que les SDD sont une particularisation des DDD ce qui permet de les employer dans le cadre théorique des chapitres 4 et 5.

Sommaire

6.1 Motivation

La réduction de la longueur des chemins s'obtient en introduisant la notion de *hiérarchie* dans les DDD. Les strates correspondant aux variables vectorielles (comme les tableaux) ou aux conteneurs (comme les files) sont déportées du DDD représentant l'état du système vers des DDD « partiels » représentant un ensemble de configurations possibles de la variable (ou conteneur) concernée. Le schéma de principe de « hiérarchisation » est donnée en figure 6.1. Le DDD de gauche (« à plat ») représente un ensemble d'états caractérisé par deux variables scalaires (a et b) et un tableau de n

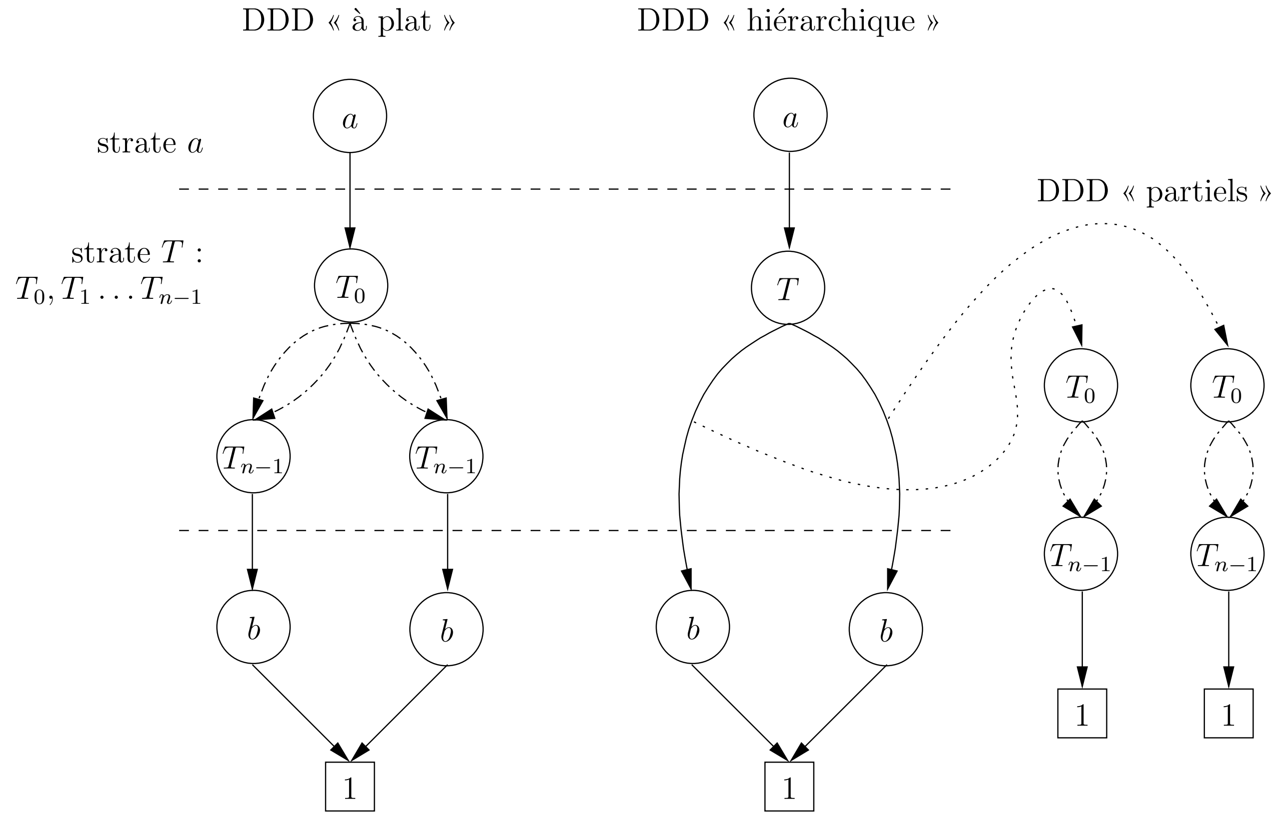


FIG. 6.1 Principe de hiérarchisation pour des variables vectorielles

cases (T). On suppose que le tableau T comporte un nombre important de cases, et que ses configurations sont nombreuses (symbolisées par des traits mixtes). Un tel cas de figure accroît la longueur et le nombre de chemins à parcourir pour atteindre la « strate b », ce qui est un des facteurs contre-productifs relevés au chapitre 5.

Dans le DDD « hiérarchique » situé au centre de la figure, on distingue le DDD principal (qui instancie les trois variables a, b et T) et les DDD « partiels ». Les configurations de T sont représentées à l'aide de DDD *qui étiquettent les arcs sortants d'un nœud étiqueté par* T. Les chemins du DDD principal menant à la strate b sont plus courts mais aussi moins nombreux (ce qui n'aurait pas été le cas en utilisant une représentation des configurations de T sur un seul niveau de profondeur).

Enfin, les deux DDD « partiels » à droite de la figure respectent le même ordre $T_0 \dots T_{n-1}$ ce qui autorise un partage *entre les étiquettes des arcs*.

La réduction de la longueur des chemins réduit les calculs intermédiaires à stocker pour atteindre les variables proches du nœud terminal 1. De plus, certains traitements peuvent être reportés sur un autre niveau de hiérarchie et ne s'appliqueront plus que sur des DDD représentant les configurations *d'une seule variable*. Par exemple, sur la figure 6.1, l'affectation $T[0] = 0$ peut être réalisée en appliquant l'homomorphisme $\langle$ setCst $\rangle$ du chapitre 2 sur les seuls DDD partiels étiquettant les arcs sortants du nœud T du DDD principal. Il y a donc moins de strates et moins de chemins à parcourir ce qui se traduit par une diminution notable du nombre de nœuds à stocker dans le cache de calcul.

6.2 Les SDD sont des DDD

Dans cette section nous explicitons le premier formalisme associé aux SDD proposé par [TM04] et montrons qu'en fait, il est inclus dans celui des DDD.

6.2.1 Définition

Les SDD définis par [TM04, CTM05] ont donné lieu à un enrichissement de la bibliothèque des DDD [DDD] d'une classe de structure en DAG dont les arcs sont étiquetés par des ensembles de valeurs. La définition de [TM04] est la suivante.

Définition 6.1 (Diagrammes de Décision d'Ensembles (SDD, [TM04])) *Soit E un ensemble de variables, pour chaque variable $e \in E$, $Dom(e)$ désigne le domaine de valeurs de e. L'ensemble $\mathbb{S}$ des SDD est défini par $d \in \mathbb{S}$ si :*
- *$d \in \{0, 1\}$ ou*
- *$d = (e, \alpha)$ avec : $e \in E$ et α est un ensemble fini de couples $c_i = (a_i, d_i)$ où $a_i \subseteq Dom(e)$, $d_i \in \mathbb{S}$ et $\forall (c_i, c_j) \in \alpha^2, i \neq j$:*
  - *$a_i \cap a_j = \emptyset$*
  - *$d_i \neq d_j$*
  - *$a_i \neq \emptyset$ et $d_i \neq 0$*

On note $e \xrightarrow{a} d$ le SDD (e, α) avec $\alpha = \{(a, d)\}$.

Cette définition laisse entrevoir le principe de l'étiquetage des arcs par des ensembles, et non par des valeurs. Elle est en fait *déduite de l'implémentation des DDD* (qui encode les fonctions α par des ensembles *finis* de couples valeur-DDD) et propose un mode d'évaluation *symbolique* de α (en l'encodant par un ensemble de couples ensembles-DDD). En somme, [TM04] suppose que la fonction $\alpha(x)$ de la définition des DDD se traduit par un ensemble fini de couples (x_i, d_i) où $x_i \subseteq Dom(e)$, $d_i \in \mathbb{D}$ et où les x_i et les d_i sont distincts deux à deux.

Or la définition des DDD ne mentionne nulle part le mode d'évaluation des fonctions α (définition 2.3) sous la forme d'un traitement indépendant de chaque valeur de $Dom(e)$. La section qui suit démontre que l'ensemble des couples (a_i, d_i) de la définition 6.1 constitue une représentation symbolique de la fonction α.

6.2.2 Construction symbolique de α

La définition 2.3 stipulant que α est telle que $\{\forall x \in Dom(e), \alpha(x) \neq 0\}$ est fini, nous allons réécrire la fonction $\alpha(x)$ sous la forme de fonction constante par morceaux :

Proposition 6.2 (Décomposition par morceaux de la fonction α) *Soit la fonction $\alpha : Dom(e) \rightarrow \mathbb{D}$ telle que $\{\forall x \in Dom(e), \alpha(x) \neq 0\}$ est fini. Il existe un unique ensemble ordonné $\{c_1 \ldots c_n\}$ composé de couples $c_i = (a_i, d_i)$ où $a_i \subseteq Dom(e)$ et $d_i \in \mathbb{D}$ qui vérifie :*
- *$\forall i, (x \in a_i \Rightarrow \alpha(x) = d_i)$*
- *$\forall (i, j), i \neq j \Rightarrow ((a_i \cap a_j = \emptyset) \wedge d_i \neq d_j)$*

On a d'autre part :$(\forall i, x \notin a_i) \Rightarrow \alpha(x) = 0$.

***Démonstration** : On part de l'ensemble des valeurs de $im_\alpha = \{\alpha(x), x \in Dom(e), \alpha(x) \neq 0\}$. Comme $\{\forall x \in Dom(e), \alpha(x) \neq 0\}$ est fini, il existe un unique ensemble ordonnant les $n = rang(\alpha)$*

valeurs que peut prendre $\alpha(x)$. On note cet ensemble $\{d_1 \dots d_n\} \in (im_\alpha)^n$ qui vérifie $\forall(i,j), i \neq j, d_i \neq d_j$ et $\forall x \in Dom(e), \alpha(x) \neq 0, \exists d_i, \alpha(x) = d_i$. A partir de chaque d_i de $\{d_1 \dots d_n\}$ on construit l'ensemble des couples $c_i = (\{x|\alpha(x) = d_i\}, d_i)$. On a alors

$$\begin{array}{rcl} x \in a_i & \Rightarrow & x \in \{x|\alpha(x) = d_i\} \\ & \Rightarrow & \alpha(x) = d_i \\ \textit{et} & & \\ a_i \cap a_{j\neq i} & = & (x|\alpha(x) = d_i) \cap (x|\alpha(x) = d_{j\neq i}) \\ & = & (x|\alpha(x) = d_i) \cap (x|\alpha(x) = d_j \neq d_i) \\ & = & \emptyset \\ \text{et} & & \\ \forall i, x \notin a_i & \Rightarrow & \nexists d_i, \alpha(x) = d_i \\ & \Rightarrow & x \notin \{\forall x \in Dom(e)|\alpha(x) \neq 0\} \\ & \Rightarrow & \alpha(x) = 0 \end{array}$$

L'ensemble des couples (a_i, d_i) ainsi construit est unique et vérifie bien les propriétés citées. □

A partir de ces couples ensembles-DDD, on construit la fonction $\alpha(x)$:

Proposition 6.3 (Construction symbolique de la fonction α**)** *Soit la fonction $\alpha : Dom(e) \rightarrow \mathbb{D}$ telle que $\{\forall x \in Dom(e), \alpha(x) \neq 0\}$ est fini. Alors il existe une décomposition canonique $\alpha(x) = \sum_{i=1}^{n} d_i \chi_i(x)$ où :*
- *$\chi_i(x)$ est la fonction caractéristique de $a_i \subseteq Dom(e)$ ($x \in a_i \Leftrightarrow \chi_i(x) = 1$) et $x \in a_i \Rightarrow \alpha(x) = d_i$.*
- *$\forall(i,j), i \neq j \Rightarrow ((a_i \cap a_j = \emptyset) \wedge (d_i \neq d_j))$.*
- *$(\exists n \in \mathbb{N}), \bigcup_1^n a_i \subset Dom(e)$ et $\alpha(x) = 0 \Leftrightarrow x \notin \bigcup_1^n a_i$.*

La dernière équivalence est la contraction des deux implications $(x \notin \bigcup_1^n a_i) \Rightarrow (\alpha(x) = 0)$ et $(x \in \bigcup_1^n a_i) \Rightarrow (\alpha(x) \neq 0)$.

***Démonstration** : On reprend la construction de la propriété 6.2 et on considère l'ensemble des n couples (a_i, d_i) obtenus.*

D'une part, les deux implications $x \notin \bigcup_1^n a_i \Rightarrow \alpha(x) = 0$ et $x \in \bigcup_1^n a_i \Rightarrow \alpha(x) \neq 0$ sont les conséquences directes de $\forall i, (x \in a_i \Rightarrow \alpha(x) = d_i)$ et $(\forall i, x \notin a_i) \Rightarrow \alpha(x) = 0$ données par la proposition 6.2.

D'autre part, $\alpha(x) = \sum_i d_i \chi_i(x)$ se démontre en isolant les cas $\alpha(x) = 0$ et $\alpha(x) \neq 0$:
- *$\alpha(x) = 0$, alors, $x \notin \bigcup_1^n a_i$ et $\forall i \in \{1 \dots n\}, x \notin a_i$ donc $\sum_i d_i \chi_i(x) = 0$.*
- *$\alpha(x) \neq 0$, $\exists! k, x \in a_k$, donc $\sum_i d_i \chi_i(x) = d_k \chi_k(x) = d_k$.*

L'ensemble des couples $c_i = (a_i, d_i)$ défini dans la propriété 6.2 étant unique pour une fonction α donnée, la décomposition $\alpha(x) = \sum_{i=1}^{n} d_i \chi_i(x)$ est unique. □

La définition 2.3 des DDD au chapitre 2 mentionne que le support de la fonction α ($\{x \in Dom(e), \alpha(x) \neq 0\}$) est fini. Cette propriété implique que l'image de la fonction α (le nombre d'arcs sortants des nœuds non terminaux) est finie. C'est cette conséquence qui est utilisée dans *toutes* les démonstrations de ce chapitre ainsi que celles de [CEPA+02]. Le support fini de la fonction α « originale » des DDD n'est donc nécessaire *qu'au sens de l'implémentation*, pour un étiquetage par valeur des arcs. La définition des SDD lève donc cette contrainte et confère une expressivité plus importante aux DDD. Dans le cadre de cette thèse, *toutes les variables ont un domaine fini* et il suffit d'imposer un support fini aux fonctions caractéristiques χ pour écarter toute ambiguïté. Néanmoins, il serait souhaitable de reprendre la définition de α pour les DDD de [CEPA+02] sous la forme « α est une fonction de $Dom(e)$ dans $\mathbb{D}$ telle que $\{\forall x \in Dom(e), \alpha(x) \neq 0\}$ est fini », et de considérer que les DDD de la bibliothèque [DDD] en sont une particularisation.

Le principe de considérer la fonction α comme une fonction constante par morceaux ne change rien à la définition des DDD (l'image de α est finie). L'étiquetage symbolique permet de s'affranchir de la représentation « valeur-DDD ». Il suggère aussi une mise en œuvre d'opérateurs (opérations ensemblistes, concaténation et homomorphismes) qui puissent tirer parti d'une représentation « ensemble-DDD » plutôt que « valeur-DDD » On conservera par la suite la notation $\alpha(x) = \sum_i d_i \chi_i(x)$ basée sur les couples (a_i, d_i).

6.3 Evaluation symbolique des opérateurs sur DDD

Nous venons de démontrer que les SDD implémentent un mode de représentation symbolique de la fonction α associée aux nœuds non terminaux des DDD. Les DDD (hiérarchiques ou non) étiquetant les arcs sortants du nœud (e, α) représentent des ensembles de configurations de la variable, c'est-à-dire les ensembles a_i des couples (a_i, α_i). La différence d'implémentation entre SDD et DDD s'accompagne, dans la bibliothèque [DDD], de nouveaux opérateurs inductifs (pour les opérations ensemblistes et les homomorphismes) adaptés à l'étiquetage des arcs par des DDD.

L'objectif de cette section est double :

1. Présenter les opérateurs sur SDD, introduits par [TM04], fournis dans la bibliothèque [DDD] et que nous utiliserons au chapitre 7.
2. Justifier ces opérateurs en ne nous basant que sur les définitions originales des DDD (chapitre 2) et sur les propositions 6.2 et 6.3, afin de conserver le cadre théorique établi aux chapitres 4 et 5.

6.3.1 Opérations ensemblistes

Le fait de réécrire la fonction α sous la forme de fonction constante par morceaux nous permet de conserver les opérateurs d'addition, de différence et d'intersection de la définition 2.4 du chapitre 2. On propose ici une réécriture de ceux-ci sous la forme ensembliste. Dans la définition 2.4 (opérations ensemblistes), on fait intervenir une application sur les arcs $\diamond \in \{+, \times, \setminus\} : Dom(e_1) \to \mathbb{D}$ avec $\forall x \in Dom(e_1)$, $(\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$. La fonction α étant constante par morceaux, on généralise l'opérateur $\diamond$ aux ensembles :

Proposition 6.4 (Evaluation symbolique des opérateurs ensemblistes) *Soient deux DDD,* $d_1 = (e_1, \alpha_1)$ *et* $d_2 = (e_2, \alpha_2)$ *tels que* $e_1 = e_2$, $\alpha_1(x) = \sum_{i=1}^{m} d_{1i}\chi_{1i}(x)$ *et* $\alpha_2(x) \sum_{j=1}^{n} d_{2j}\chi_{2j}(x)$. *Les opérateurs d'union, d'intersection et de différence entre* d_1 *et* d_2 *peuvent être symboliquement éva-*

lués à l'aide des équations suivantes :

$$\begin{aligned}
(\alpha_1 + \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \left(\begin{array}{lrl} & (d_{1i} + d_{2j}) & (\chi_{1i}(x) \times \chi_{2j}(x)) \\ + & d_{1i} & (\chi_{1i}(x) - \chi_{2j}(x)) \\ + & d_{2j} & (\chi_{2j}(x) - \chi_{1i}(x)) \end{array} \right) \\
(\alpha_1 \times \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} (d_{1i} \times d_{2j})(\chi_{1i}(x)\chi_{2j}(x)) \\
(\alpha_1 \backslash \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \left(\begin{array}{lrl} & (d_{1i} \backslash d_{2j}) & (\chi_{1i}(x) \times \chi_{2j}(x)) \\ + & d_{1i} & (\chi_{1i}(x) - \chi_{2j}(x)) \end{array} \right)
\end{aligned}$$

Démonstration *: La démonstration est donnée en annexe page 129. Elle consiste à évaluer les opérateurs $\diamond \in \{+, \times, \backslash\}$ en tenant compte de la décomposition par morceaux des fonctions α_1 et α_2.*
$\square$

En pratique les opérations portant sur les fonctions caractéristiques (χ) sont réalisées à l'aide des opérations ensemblistes sur les ensembles a_i, à savoir les mêmes opérateurs $\diamond \in \{+, \times, \backslash\}$ sur DDD mais appliqués sur un niveau de hiérarchie inférieur.

Grâce à ce mode de réécriture des fonctions α, on aura autant de couples ensemble-SDD que de sous-DAG, et on voit que les opérateurs peuvent être évalués à partir de la représentation de ces ensembles pourvu qu'ils soient muni d'opérateurs d'union, d'intersection et de différence.

Les équations ci-dessus permettent donc d'évaluer les opérateurs ensemblistes de la définition 2.4 en tirant partie de l'étiquetage symbolique des arcs d'un DDD.

6.3.2 Homomorphismes et Concaténation

Jusqu'ici, le fait de considérer les fonctions α d'un point de vue symbolique plutôt qu'explicite ne change rien aux propriétés des DDD. En revanche, la définition 2.8 des homomorphismes inductifs sur DDD fait explicitement mention des couples valeur-DDD. Comme nous souhaitons conserver l'étiquetage symbolique des arcs, nous proposons de réécrire la définition des homomorphismes en faisant apparaître la représentation symbolique de α :

Proposition 6.5 (Homomorphismes inductifs sur espaces symboliques) *Soient c un SDD et $\phi(e, x)_{e \in E, x \in Dom(e)}$ une famille d'homomorphismes.*

$$\Phi(d) = \begin{cases} 0 & \text{si } d = 0 \\ \top & \text{si } d = \top \\ c & \text{si } d = 1 \\ \sum_{i=1}^{n} \Phi(e, a_i)(d_i \chi_i(x)) & \text{si} \left\{ \begin{array}{rcl} d &=& (e, \alpha) \\ \alpha(x) &=& \sum_{i=0}^{n} (d_i \chi_i(x)) \\ \Phi(e, a_i) &=& \sum_{x \in a_i} \phi(e, x) \end{array} \right. \end{cases}$$

est un homomorphisme.

Démonstration *: La démonstration est donnée en annexe page 130. Elle consiste à développer l'expression de $\sum_{i=1}^{n} \Phi(e, a_i)(d_i \chi_i(x))$ puis de la réduire pour obtenir $\sum_{x \in Dom(e)} \phi(e, x)\alpha(x)$ (définition 2.8).* $\square$

Il est donc possible de réaliser des homomorphismes inductifs définis localement sur des couples variable-*ensemble* à condition de disposer d'homomorphismes sur ces ensembles. On conservera la notation $\Phi(e, a_i)$ pour la définition d'homomorphismes inductifs (on supposera donc implicitement que $\alpha(x) = \sum_{i=0}^{n}(d_i\chi_i(x))$.

L'opérateur de concaténation de la définition 2.5 doit lui aussi être réécrit pour être compatible avec l'étiquetage symbolique :

Proposition 6.6 (Concaténation avec étiquetage symbolique) *Soient deux DDD d_1 et d_2, le DDD $d_1 \odot d_2$ défini comme suit :*

$$d_1 \odot d_2 = \begin{cases} 0 & \text{si } d_1 = 0 \vee d_2 \equiv 0 \\ d_2 & \text{si } d_1 = 1 \\ \top & \text{si } d_1 = \top \wedge d_2 \not\equiv 0 \\ (e, \sum_{i=1}^{n}(d_{1i} \odot d_2)\chi_{1i}(x)) & \text{si } d_1 = (e, \alpha) \\ & \alpha(x) = \sum_{i=0}^{n}(d_{1i}\chi_{1i}(x)) \end{cases}$$

est le résultat de la concaténation $d_1 \cdot d_2$.
***Démonstration** : La démonstration est donnée en annexe page 131.* □

6.4 Conclusion

Nous avons montré que l'apport de la hiérarchie dans les DDD permettait

- de réduire la longueur des chemins sur le DAG dont l'effet contre-productif a été relevé au chapitre 5,
- d'accroître les capacités de partage de la structure par l'introduction de DDD « partiels »,
- de reporter certains opérateurs inductifs sur ces derniers.

Les Diagrammes de Décision d'Ensembles (SDD), proposés par Jean-Michel Couvreur et Yann Thierry-Mieg dans [TM04, CTM05] permettent de représenter ces DDD hiérarchiques. Nous avons démontré que les SDD et leurs opérateurs sont inclus dans le formalisme des DDD, ils peuvent donc être utilisés dans le cadre théorique défini aux chapitres 4 et 5. Néanmoins, les différences d'implémentation et l'apparition de la hiérarchie imposent de préciser un mode d'accès à toute donnée quel que soit son niveau de hiérarchie.

Chapitre 7

Approche Promela-SDD

Nous avons montré au chapitre 6 que les SDD implémentent des DDD avec un étiquetage symbolique des arcs. L'objectif de ce chapitre est de montrer comment réutiliser les méthodes mises en œuvre au chapitre 5 en tenant compte de la différence d'implémentation entre DDD et SDD. Les points développés sont les suivants :
- la représentation d'états en exploitant la hiérarchie du programme Promela,
- les mécanismes pour rendre utilisables les homomorphismes inductifs lorsqu'ils s'appliquent sur des ensembles de valeurs et le passage d'un niveau de hiérarchie à un autre,
- la mise en œuvre des techniques de saturations décrites au chapitre 2.

Cette étude a conduit à l'élaboration d'un nouveau prototype de model-checker dont nous comparons les performances avec le premier.

Sommaire

7.1 Représentation hiérarchique des états

Aux chapitres 4 et 5, le DDD représentant l'état initial d'un système était obtenu en construisant la configuration initiale de chacun de ses éléments (variables partagées, canaux et processus) dont les DDD étaient concaténés. Certaines variables sont encodées sur des DDD dont la longueur des chemins excède un nœud : il s'agit des variables vectorielles, des conteneurs et des processus. Le principe de représentation hiérarchique consiste à regrouper ces nœuds supplémentaires en un seul DDD qui étiquettera l'arc sortant du nœud étiqueté par la variable concernée. Pour ce faire, on s'appuie sur les trois éléments qui caractérisent la déclaration d'une variable Promela :

– le type de la variable (entière ou structurée),
– le nom de la variable,
– le mode de déclaration (variable scalaire, tableau ou canal).

On suppose ici que la valeur d'initialisation d'une variable entière est toujours nulle : notre prototype tient compte des variables préinitialisées, mais au prix de complications qui n'ont pas d'utilité dans cette section. La représentation hiérarchique est obtenue en n'utilisant, sur le premier niveau de hiérarchie, *qu'un seul nœud par variable globale* (dont les identificateurs de processus). A chaque type est associé un DDD d'initialisation. Ce DDD vient étiqueter l'arc sortant du nœud étiqueté par la variable instanciée dans le DDD de configuration initiale. Pour les variables nécessitant plusieurs nœuds (structures, processus, files etc.) ces nœuds sont déportés vers un nouveau niveau de hiérarchie.

On applique les règles suivantes :

1. A chaque type est associé un DDD d'initialisation noté ddd_{type}.
2. Pour les valeurs entières, le DDD d'initialisation est $ddd_{int} = I \xrightarrow{0} 1$.
3. La configuration initiale d'une variable scalaire (nommée nom) est donnée par le DDD $nom \xrightarrow{ddd_{type\_nom}} 1$. Ainsi la déclaration `int a;` aboutit au DDD $a \xrightarrow{I \xrightarrow{0} 1} 1$.
4. un type structuré comportant n champs f_i a le DDD d'initialisation suivant : $f_1 \xrightarrow{ddd_{type\_f_1}} f_2 \xrightarrow{ddd_{type\_f_2}} \dots f_n \xrightarrow{ddd_{type\_fn}}$.
5. La configuration initiale d'un tableau de taille n est similaire à celle d'une structure comportant autant de champs du même type. Ainsi la déclaration `type t[3];` aboutit au DDD $t \xrightarrow{2 \xrightarrow{ddd_{type}} 1 \xrightarrow{ddd_{type}} 0 \xrightarrow{ddd_{type}} 1} 1$.
6. Les canaux étant initialement vides, leur DDD d'initialisation est toujours le même : $C \xrightarrow{taille} C \xrightarrow{\sharp} 1$. Ainsi la déclaration `chan q = [2] of {int}` aboutit au DDD $q \xrightarrow{C \xrightarrow{2} C \xrightarrow{\sharp} 1} 1$, mais les DDD correspondant aux valeurs insérées dans `q` devront être compatibles avec ddd_{int}.
7. La configuration initiale d'un processus est obtenue par la concaténation de la variable entière PC suivie par les variables locales : $PC \xrightarrow{I \xrightarrow{0} 1} local_1 \xrightarrow{ddd_1} \dots$.
8. La déclaration d'un processus p est donnée par le DDD $p \xrightarrow{ddd_p} 1$ où ddd_p est le DDD d'initialisation de p.
9. La configuration initiale d'un programme est donnée par la concaténation des DDD de déclaration des variables partagées, des canaux et des processus suivant un ordre défini par l'utilisateur (par défaut celui d'apparition dans le programme Promela).

Exemple 7.1 (Tableaux de points sur le plan discret) *La figure 7.1 donne le code Promela de l'instanciation de points du plan discret et le DDD résultant : On remarque tout d'abord que l'accès aux différentes variables sur le DDD principal est considérablement amélioré : l'accès aux valeurs de* `t2` *ne nécessite plus de traverser les 6 nœuds qui auraient été utilisés pour encoder* `t1` *sur une structure « à plat ». De plus, quels que soient leurs niveaux de hiérarchie, les nœuds de DDD identiques sont partagés : le DDD étiquetant l'arc issu du nœud* `p` *est identique à ceux qui sont rangés dans les deux tableaux, lesquels partagent par ailleurs le DDD* ddd_{t2}.

```
typedef plot {
   int x;
   int y;
};

plot p;
plot t1[3];
plot t2[2];
```

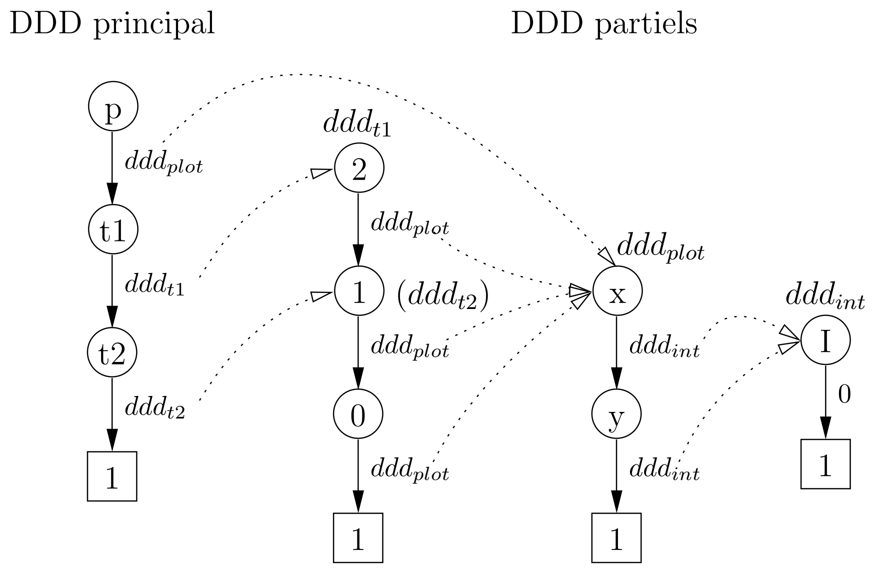


FIG. 7.1 Représentation de points du plan discret

Exemple 7.2 (Un SDD initial complet) *Considérons le programme suivant inspiré des exemples du chapitre 3 :*

```
typedef Header {
   int source;
   int dest;
   int id;
}
typedef Message{
   Header head;
   int data[2];
}

typedef Acknowledgement {
   bool status;
   int id;
}

chan StoR = [3] of Message;
chan RtoS = [1] of Acknowledgement;

active[1] proctype Sender(){
Message Mess;
Acknowledgement Ack;
/* Instructions */
}

active[1] proctype Receiver(){
```

```
Message Mess;
Acknowledgement Ack;
/* Instructions */
}
```

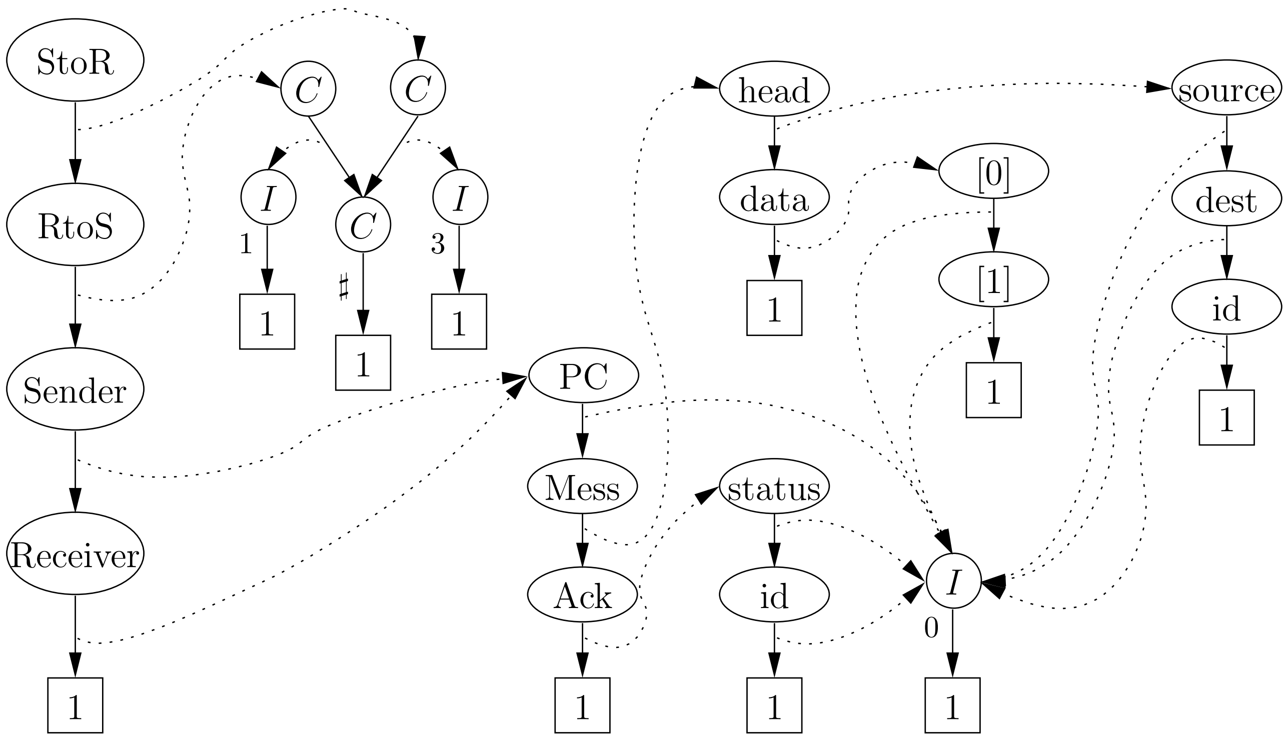


FIG. 7.2 DDD pricipal et partiels d'un état initial

La figure 7.2 donne l'ensemble des DDD créés pour l'instanciation du DDD initial (à l'extrême gauche de la figure). Seuls les nœuds terminaux 1 *sont représentés plusieurs fois par souci de lisibilité. Le DDD principal a une longueur de chemin fixe de 4 nœuds contre 20 dans le cas d'un DDD « à plat », cette valeur pouvant atteindre 37 nœuds lorsque tous les canaux sont pleins. De plus, processus et canaux utilisent les mêmes types structurés ce qui améliore grandement le partage.*

7.2 Implémentation hiérarchique des homomorphismes

D'un point de vue formel, il n'y a pas de différence entre les DDD et les SDD. La différence entre les deux structures est liée à l'implémentation des DDD, et nous avons vu au chapitre 6 que les opérateurs associés aux SDD diffèrent aussi de ceux des DDD car ils manipulent des ensembles d'états. Nous avons montré que ces derniers sont formellement équivalents à ceux des DDD « à plat » mais se pose le problème de *l'accès aux valeurs* par les homomorphismes :

- Les arcs représentant un *ensemble de valeurs*, il est nécessaire de différencier ces valeurs si l'homomorphisme est sensible à la variable représentée.
- Plusieurs variables pouvant être instanciées sur un même DDD partiel, il faut se munir d'une méthode permettant d'accéder aux bonnes variables, indépendamment des autres.

Dans cette section, nous montrons comment résoudre ces deux problèmes pour adapter les homomorphismes du chapitre 5 aux DDD hiérarchiques.

7.2.1 Différenciation des valeurs : Split

Dans le cas d'un SDD, tout arc sortant d'un nœud représente un ensemble de valeurs. Supposons que nous voulons appliquer l'homomorphisme $\langle\ \mathsf{setExpr}\ (b, a)\ \rangle$ sur le DDD $a \xrightarrow{\{1,2\}} b \xrightarrow{\{0\}} 1$. La propagation de cet homomorphisme devra être différente suivant *chacune* des valeurs que prendra a, en effet, celles-ci seront individuellement affectées à b, et le résultat devrait être :

$$
\begin{aligned}
& \langle\ \mathsf{setExpr}\ (b,a)\ \rangle a \xrightarrow{\{1,2\}} b \xrightarrow{\{0\}} 1 \\
= \ & \langle\ \mathsf{setExpr}\ (b,a)\ \rangle \begin{pmatrix} & a \xrightarrow{\{1\}} b \xrightarrow{\{0\}} 1 \\ + & a \xrightarrow{\{2\}} b \xrightarrow{\{0\}} 1 \end{pmatrix} \\
= \ & \begin{pmatrix} & a \xrightarrow{\{1\}} \langle\ \mathsf{setCst}\ (b,1)\ \rangle b \xrightarrow{\{0\}} 1 \\ + & a \xrightarrow{\{2\}} \langle\ \mathsf{setCst}\ (b,2)\ \rangle b \xrightarrow{\{0\}} 1 \end{pmatrix} \\
= \ & \begin{pmatrix} & a \xrightarrow{\{1\}} b \xrightarrow{\{1\}} 1 \\ + & a \xrightarrow{\{2\}} b \xrightarrow{\{2\}} 1 \end{pmatrix}
\end{aligned}
$$

Le procédé qui a permis d'identifier chacune des valeurs de a s'appelle « Split » et n'est pas fourni dans la bibliothèque [DDD]. Il s'agit d'un procédé qui, pour un ensemble de n valeurs renvoie n singletons le long desquels les homomorphismes se propagent de façon différenciée. Les ensembles étant représentés sur des SDD (au sens de la bibliothèque), ou des DDD comportant une unique strate I, l'algorithme de principe de la fonction $Split$ appliquée sur un SDD est défini comme suit :

Algorithme 7.1 : Différentiation de tous les chemins d'un SDD : Split

Input : Un SDD S
Output : Un ensemble de SDD ret
Data : Un ensemble de SDD fils $sons$
Data : Un SDD fils son
Data : Un SDD intermédiaire val
Data : Un ensemble de SDD intermédiaires $vals$

$ret \leftarrow \emptyset$
if $(S = 0) \vee (S = 1) \vee (S = \top)$ **then**
    `Les nœuds terminaux sont inchangés. Comme au chapitre 4, les` $\top$ `ne doivent pas apparaître.`
    $ret \leftarrow S$
else
    S `s'écrit` $S = (e, \alpha)$ `où` $\alpha(x) = \sum_1^n d_i \chi_i(x)$ `est représentée sous la forme de couples` (a_i, d_i)
    `L'algorithme s'applique à deux niveaux de hiérarchie différents : Les ensembles de valeurs étiquetant les arcs sortants et les nœuds vers lesquels ces arcs sont dirigés.`
    for $i = 1$ **to** n **do**
        $sons = Split(d_i)$
        foreach $son \in sons$ **do**
            $vals \leftarrow Split(a_i)$
            foreach $val \in vals$ **do**
                $ret \leftarrow ret \cup e \xrightarrow{val} son$
return ret

Split énumère toutes les valeurs d'un ensemble *fini* de valeurs ou de configurations représenté sur un DDD hiérarchique. Pour chaque arc sortant d'un nœud intermédiaire, étiqueté par a_i et dirigé vers d_i, on dispose de chacune des valeurs $x \in a_i$, et l'évaluation de l'homomorphisme sur DDD hiérarchique procède alors suivant le même mécanisme que pour un DDD « à plat ». Cette propriété n'intervient que *localement* : les DDD « fils » sont inchangés et conservent leur représentation hiérarchique. Aussi, il ne doit être appliqué que lorsque l'on désire énumérer, à un niveau de hiérarchie donné, toutes les valeurs d'une variable dans un (ou plusieurs) ensemble(s) des valeurs.

Exemple 7.3 (Split) *La figure 7.3 donne un exemple d'utilisation de l'algorithme de différenciation. Il correspond à l'affectation* `p2=p1` *où* `p1` *et* `p2` *sont des points du plan.*

Le traitement comporte deux étapes :

1. *Le nœud étiqueté par* $p1$ *est atteint, chacune de ses valeurs est nécessaire, on applique l'algorithme « Split » qui renvoie un ensemble de valeurs distinctes, chacune particularise l'homomorphisme propagé. L'algorithme « Split » n'est pas appliqué sur le SDD entier mais seulement sur l'ensemble des valeurs qui étiquette l'arc sortant de* $p1$.
2. *Il y a deux affectations constantes distinctes (*$p1$ *ne compte qu'un seul arc sortant), qui donnent lieu à deux chemins vérifiant bien* a posteriori $p1 = p2$. *On notera que l'homomorphisme propagé d'affectation constante ne porte pas sur une valeur entière mais sur un DDD.*

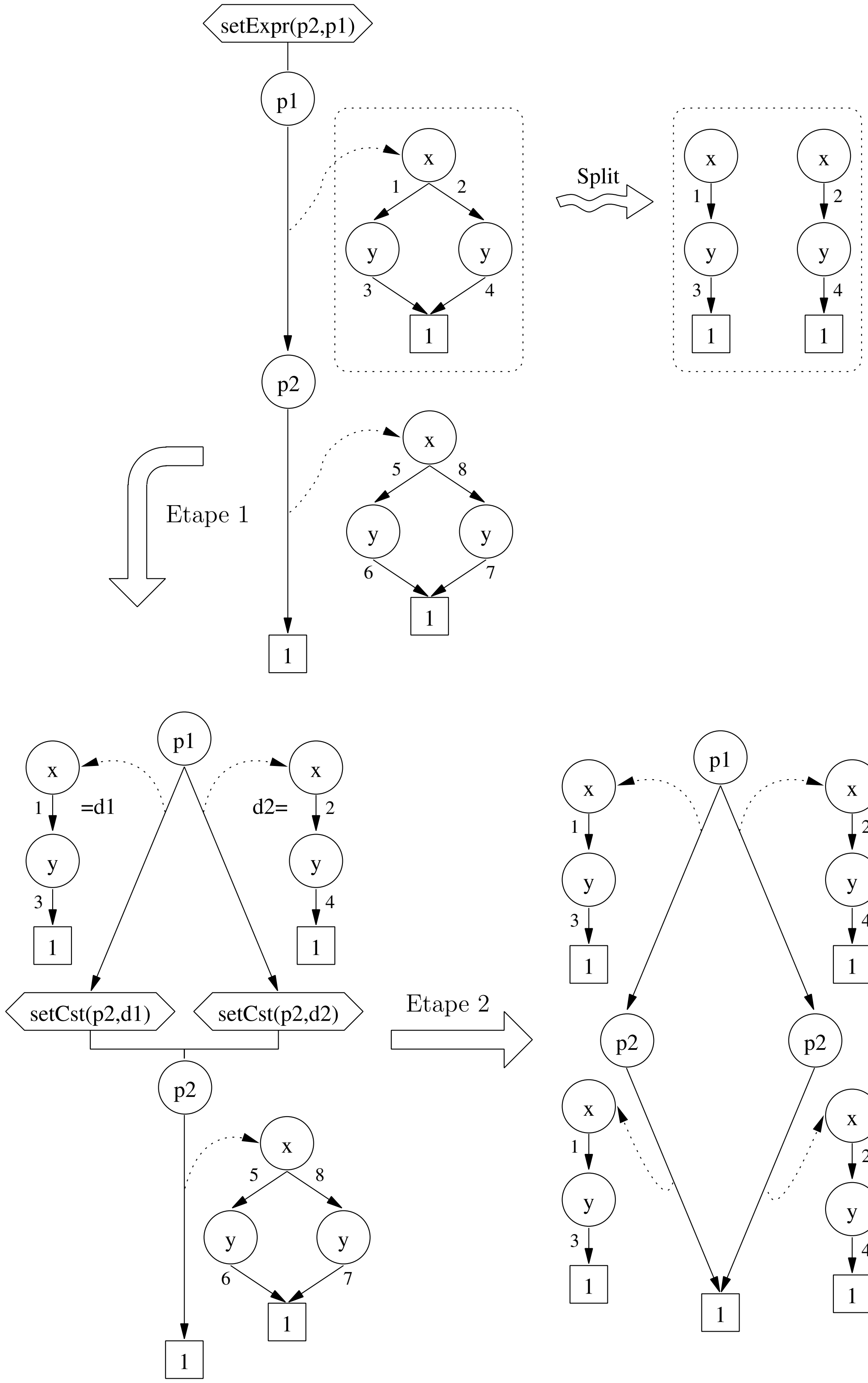


FIG. 7.3 Affectation utilisant l'algorithme « Split »

On insiste sur le fait que sans la particularisation des valeurs de $\mathrm{p}1$*, c'est-à-dire en affectant à* $p2$ *l'*ensemble *des valeurs de* $\mathrm{p}1$*, on aurait abouti à des états contradictoires avec l'affectation effectuée, comme* $\mathrm{p}1.x = 1$ *et* $p2.x = 2$*.*

7.2.2 Mise à plat et Reconstruction

La méthode de différenciation explicite de la section 7.2.1 peut s'avérer extrêmement consommatrice en ressources matérielles, et fait perdre *localement* les avantages de l'étiquetage symbolique des arcs. Dans les cas où il n'est pas nécessaire d'accéder explicitement à toutes les données qui caractérisent la grandeur d'une variable, il ne faut différencier que les valeurs utiles. Si l'on reprend l'exemple d'affectations sur les points du plan, et que l'on souhaite réaliser l'affectation `p2.x = p1.y`, les valeurs que prendra le champ `x` de `p1` n'ont pas d'influence sur le résultat mais induisent autant d'homomorphismes que `p1.x` a de valeurs différentes. Si le nœud $p1$ représente un ensemble de n points sur une droite $y = c$, l'algorithme $Split$ renverra n points alors que la seule valeur c de y nous était nécessaire. Les n homomorphismes propagés sont bien entendu identiques mais n accès au cache de calcul auront été nécessaires pour aboutir à ce résultat. La figure 7.4 montre ce que donnerait l'algorithme $Split$ dans ce cas.

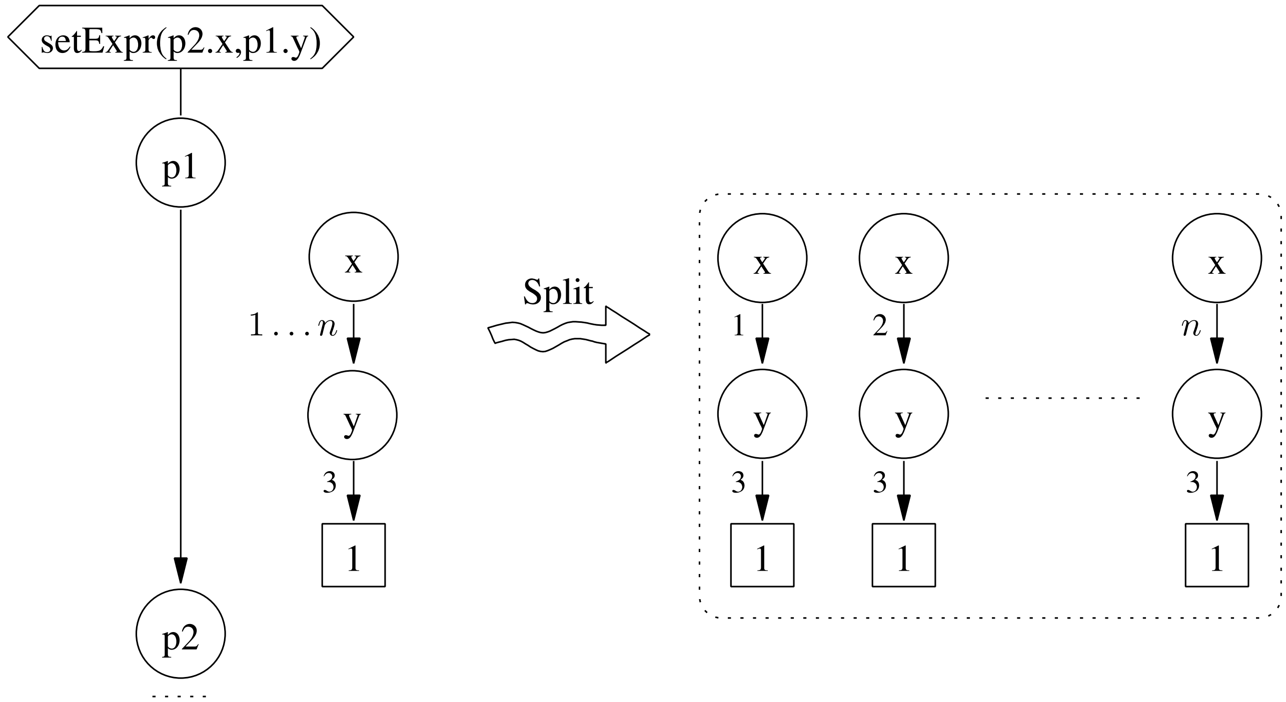


FIG. 7.4 Une mauvaise utilisation de « Split »

Pour procéder à une différenciation efficace, c'est-à-dire retardée jusqu'à ce que la valeur recherchée soit atteinte, on construit un homomorphisme ($\langle$ flat () $\rangle$) chargé d'abolir localement un niveau de hiérarchie. Le principe adopté consiste à *réintégrer* les DDD partiels au niveau de hiérarchie supérieur à l'aide de l'opérateur de concaténation. Un délimiteur de fin de bloc (noté EOB) est aussi ajouté pour rétablir le niveau de hiérarchie aboli une fois les traitements achevés :

Définition 7.1 (Homomorphisme $\langle\ \mathsf{flat}\ ()\ \rangle$) *Soit $EOB \notin E$ une variable étiquetant les délimiteurs de blocs. L'homomorphisme $\langle\ \mathsf{flat}\ ()\ \rangle$ est défini comme suit :*

$$\langle\ \mathsf{flat}\ ()\ \rangle(d) = \begin{cases} 0 & \text{si } d = 0 \\ \top & \text{si } d = \top \\ \top & \text{si } d = 1 \\ a_i \cdot EOB \xrightarrow{\sharp} \langle\ \mathsf{id}\ \rangle & \text{si } d = (e, a_i) \end{cases}$$

Cette définition n'est valable que si a_i est représenté sur un DDD bien défini.

Le niveau de hiérarchie inférieur représentant les configurations de la variable e est intégré au premier niveau de hiérarchie, lequel ne contient alors plus de nœud étiqueté par e. La variable EOB est utilisée par l'homomorphisme $\langle\ \mathsf{rebuild}\ ()\ \rangle$ pour rétablir le niveau de hiérarchie perdu :

Définition 7.2 (Homomorphisme $\langle\ \mathsf{rebuild}\ ()\ \rangle$) *Soit EOB la variable introduite par la définition 7.1. L'homomorphisme inductif $\langle\ \mathsf{rebuild}\ ()\ \rangle$ est défini comme suit :*

$$\langle\ \mathsf{rebuild}\ (var, t)\ \rangle(d) = \begin{cases} 0 & \text{si } d = 0 \\ \top & \text{si } d = \top \\ \top & \text{si } d = 1 \\ \langle\ \mathsf{rebuild}\ (var, t \cdot a_i)\ \rangle & \text{si } d = (e \neq EOB, a_i) \\ var \xrightarrow{t} \langle\ \mathsf{id}\ \rangle & \text{si } d = (e = EOB, a_i) \end{cases}$$

Cette définition n'est valable que si a_i est représenté sur un DDD bien défini.

Le rapport entre les deux homomorphismes $\langle\ \mathsf{flat}\ ()\ \rangle$ et $\langle\ \mathsf{rebuild}\ ()\ \rangle$ est le suivant :

Proposition 7.3 (Composition de $\langle\ \mathsf{flat}\ ()\ \rangle$ et $\langle\ \mathsf{rebuild}\ ()\ \rangle$) *Soit un DDD $d = (e, \alpha)$, alors*

$$\langle\ \mathsf{rebuild}\ (e, 1)\ \rangle \circ \langle\ \mathsf{flat}\ ()\ \rangle(e, \alpha) = (e, \alpha)$$

***Démonstration** : En annexe page 131.* □

L'emploi conjugué de ces deux homomorphismes permet d'annuler ponctuellement un niveau de hiérarchie, et de travailler sur deux espaces d'états en relation de correspondance biunivoque l'un (obtenu par $\langle\ \mathsf{flat}\ ()\ \rangle$) représentant l'autre (qui peut être reconstruit par $\langle\ \mathsf{rebuild}\ ()\ \rangle$) privé localement d'un niveau de hiérarchie structurelle. Ceci autorise la création d'un homomorphisme sensible aux différents chemins (hiérarchiques) du DDD *qui étiquette un arc* sans avoir à tous les différencier.

Proposition 7.4 *Soient s un type structuré et un homomorphisme inductif $\langle\ \mathsf{H}\ ()\ \rangle$ réalisant une fonction $F(s.field_1 \ldots s.field_m, x_1 \ldots x_n)$, où $s.field_i$ représente un champ de s. On construit l'homomorphisme inductif $\langle\ \mathsf{h}\ ()\ \rangle$ réalisant la fonction $f(field_1 \ldots field_m, x_1 \ldots x_n)$. Appliqué à un nœud étiqueté par s, $\langle\ \mathsf{H}\ ()\ \rangle$ peut s'évaluer comme suit :*

$$\begin{aligned} & \langle\ \mathsf{H}\ (s.field_1 \ldots s.field_m, x_1 \ldots x_n)\ \rangle(s, e_i) \\ = \ & \langle\ \mathsf{rebuild}\ (s, 1)\ \rangle \circ \langle\ \mathsf{h}\ (field_1 \ldots field_m, x_1 \ldots x_n)\ \rangle \circ \langle\ \mathsf{flat}\ ()\ \rangle(s, e_i) \end{aligned}$$

Explication *: On a vu la correspondance biunivoque entre les espaces d'états avec ou sans un niveau de hiérarchie donné (ici s). L'homomorphisme* $\langle \mathsf{h}\ () \rangle$ *est aussi en correspondance biunivoque avec* $\langle \mathsf{H}\ () \rangle$*, et réalise le même traitement sur un ensemble de variables privées de la hiérarchie liée à s, ce qui nous place dans le même formalisme que les DDD avec un étiquetage par valeurs. L'homomorphisme* $\langle \mathsf{rebuild}\ () \rangle$ *se charge de reconstruire le DDD avec la hiérarchie initiale.*

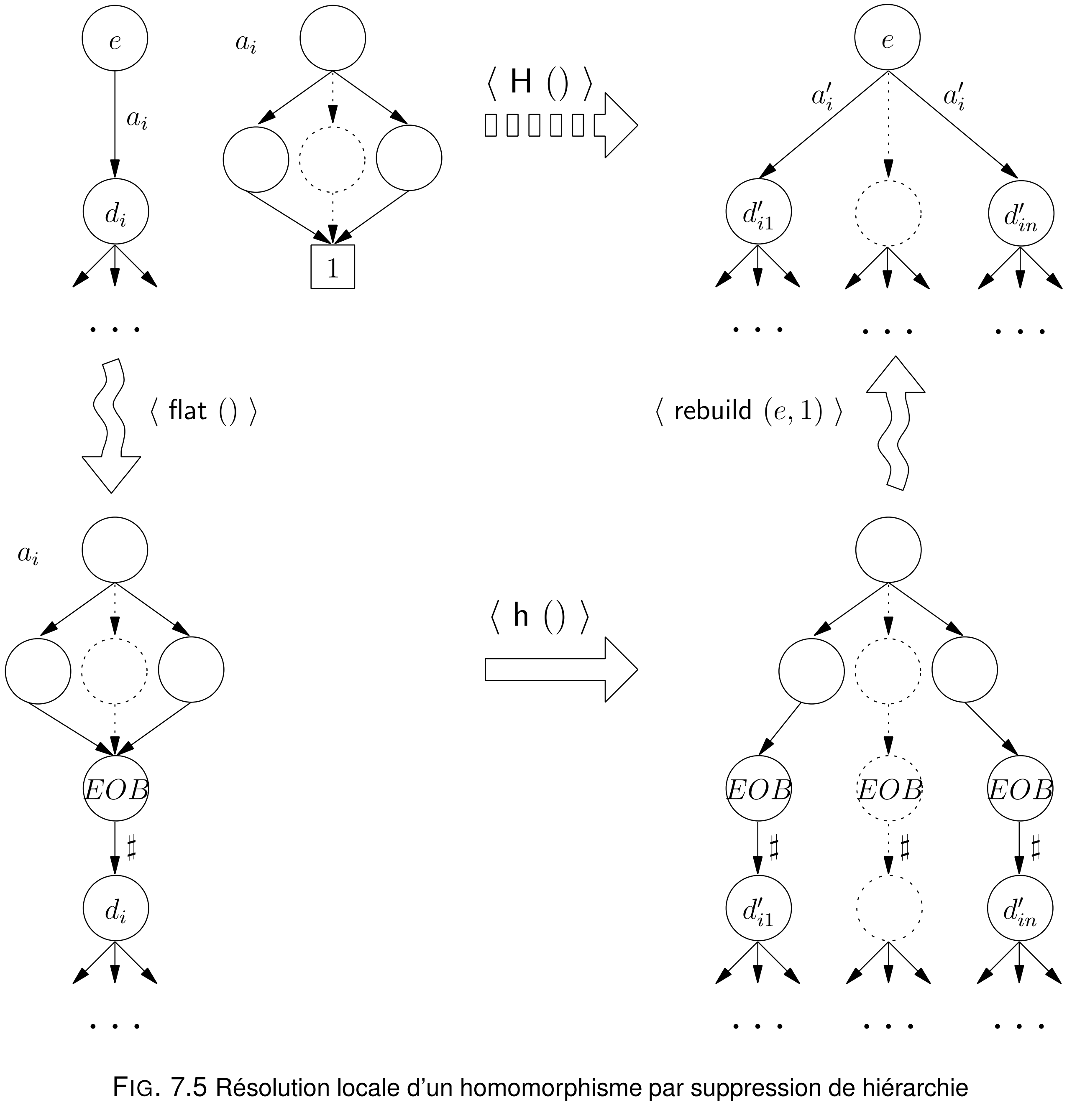


FIG. 7.5 Résolution locale d'un homomorphisme par suppression de hiérarchie

La figure 7.5 résume l'application de $\langle \mathsf{H}\ () \rangle$ *par l'intermédiaire de* $\langle \mathsf{h}\ () \rangle$ *suite aux changements de contexte (s hiérarchique ou non) provoqués par l'emploi de* $\langle \mathsf{flat}\ () \rangle$ *et* $\langle \mathsf{rebuild}\ () \rangle$*. Le SDD a_i se* *substitue à la variable e et est borné* en aval *par un nœud EOB. Le nom de la variable e est conservé par le mécanisme de reconstruction qui borne a_i* en amont. □

Les homomorphismes $\langle$ flat () $\rangle$ et $\langle$ rebuild () $\rangle$ permettent donc de naviguer d'un niveau de hiérarchie à un autre.

Proposition 7.5 (Propagation à travers plusieurs niveaux de hiérarchies) *Lorsque l'homomorphisme $\langle$ h () $\rangle$ ne se propage pas sur les strates inférieures à celle de la variable mise à plat (i.e. $\forall i, \langle \mathsf{h}\ () \rangle(a_i \cdot EOB \xrightarrow{\sharp} d_i) = \langle \mathsf{h}\ () \rangle(a_i) \cdot EOB \xrightarrow{\sharp} d_i$), l'utilisation des homomorphismes $\langle$ flat () $\rangle$ et $\langle$ rebuild () $\rangle$ peut être remplacée par une application de $\langle$ h () $\rangle$ sur les DDD étiquetant les arcs sortants.*
***Démonstration** :*

$$\begin{aligned}
\langle \mathsf{H}\ () \rangle(e, a_i) &= \langle \mathsf{rebuild}\ (e,1) \rangle \circ \langle \mathsf{h}\ () \rangle \circ \langle \mathsf{flat}\ () \rangle(e, a_i) \\
&= \langle \mathsf{rebuild}\ (e,1) \rangle \circ \langle \mathsf{h}\ () \rangle(a_i \cdot EOB \xrightarrow{\sharp} d_i) \\
&= \langle \mathsf{rebuild}\ (e,1) \rangle \circ \langle \mathsf{h}\ () \rangle(a_i) \cdot EOB \xrightarrow{\sharp} d_i \\
&= \langle \mathsf{rebuild}\ (e,1) \rangle \circ \langle \mathsf{flat}\ () \rangle e \xrightarrow{\langle \mathsf{h}\ () \rangle(a_i)} d_i \\
&= e \xrightarrow{\langle \mathsf{h}\ () \rangle(a_i)} d_i
\end{aligned}$$

On note que, dans le cadre de cette étude, a_i étant un DDD, l'évaluation paresseuse de l'étiquette de l'arc résultant est possible. □

Corollaire 7.6 (Affectations hiérarchiques de constantes) *L'homomorphisme d'affectation d'une constante à un champ d'une structure s se propagera* uniquement dans *les étiquettes des arcs sortants d'un nœud étiqueté par s :*

$$\langle \mathsf{setCst}\ (s.field, c) \rangle(s, a_i) \quad = \quad s \xrightarrow{\langle \mathsf{setCst}\ (field,c) \rangle a_i} d_i$$

L'exemple qui suit résume le principe de mise en œuvre d'homomorphismes sur des DDD hiérarchiques :

Exemple 7.4 (Affectation par un champ d'une structure) *Dans la figure 7.6 on montre comment effectuer l'affectation `p2.x=p1.y` en ignorant la multiplicité des valeurs de `p1.x`. Les DDD utilisés pour représenter les ensembles de valeurs entières ne figurent pas par souci de lisibilité. Les étapes de calcul (indiquées par des flèches numérotées) sont les suivantes :*

1. *Suppression de la hiérarchie de $p1$ par l'homomorphisme $\langle$ flat () $\rangle$. La balise EOB (utile à l'homomorphisme $\langle$ rebuild () $\rangle$) est placée, l'homomorphisme associé à $\langle$ setExpr $(p2.x, p1.y)$ $\rangle$ devient $\langle$ setExpr $(p2.x, y)$ $\rangle$.*
2. *Propagation de $\langle$ setExpr $(p2.x, y)$ $\rangle$ à travers les champs de $p1$. Le passage à travers la strate x est indépendant du nombre de valeurs que prend x (cet ensemble de valeurs entières est encodé sur un seul DDD qui n'est pas représenté sur la figure). Après application de l'algorithme Split sur y (et y seulement), l'homomorphisme $\langle$ setExpr $(p2.x, y)$ $\rangle$ devient une affectation constante $\langle$ setCst $(p2.x, 3)$ $\rangle$ (s'il y avait eu plusieurs valeurs possibles de y, on aurait appliqué Split sur les noeuds étiquetant les arcs sortants de y, et nulle part ailleurs).*

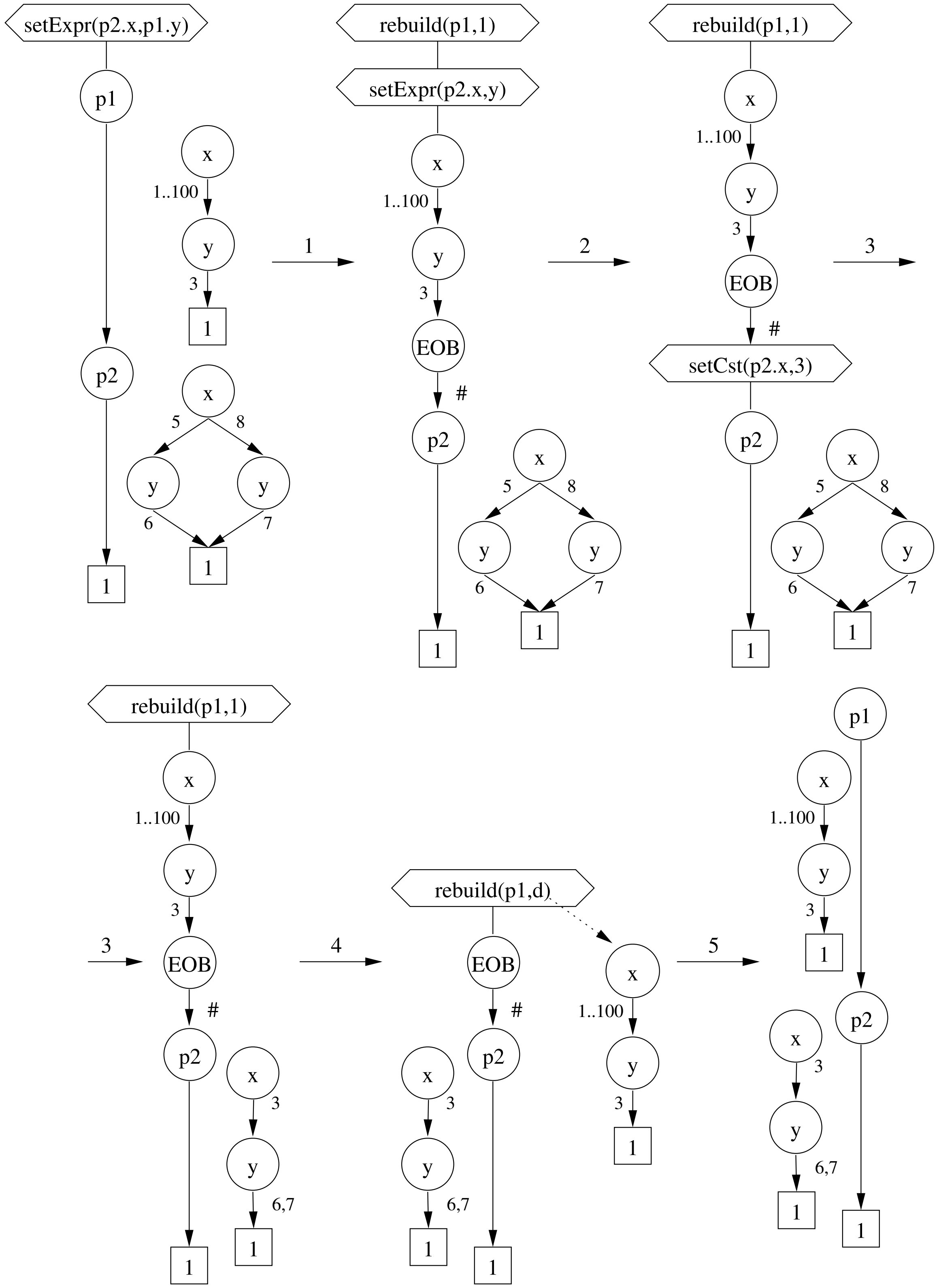


FIG. 7.6 Mise à plat et reconstruction de SDD

3. *L'affectation de valeur constante sur le nœud $p2$ se fait en appliquant le corollaire 7.6.*
4. *L'homomorphisme* $\langle$ rebuild () $\rangle$ *collecte les valeurs de x et y (dans le contexte privé de la hiérarchie liée à $p1$) et construit le DDD d des valeurs de p_1.*
5. *Arrivé à la balise EOB, l'homomorphisme* $\langle$ rebuild () $\rangle$ *reconstruit la structure $p1$ à l'aide du DDD d.*

La méthode employant $\langle$ flat () $\rangle$ et $\langle$ rebuild () $\rangle$ permet d'accéder à des valeurs encodées dans une structure donnée. Il nous reste à montrer comment cette méthode est employée en terme général au sein d'un homomorphisme quelconque. Au chapitre 5 nous avons vu que l'homomorphisme d'affectation d'expressions faisait intervenir les principaux traitements élémentaires (comme $\langle$ up () $\rangle$ et $\langle$ downExpr () $\rangle$) à partir desquels étaient dérivés tous les homomorphismes nécessaires à la construction de l'opérateur $post$. Dans la section suivante, nous montrons comment adapter l'homomorphisme $\langle$ setExpr () $\rangle$ aux DDD hiérarchiques (la section 7.2.4 est consacrée à l'opérateur pre).

7.2.3 Application aux affectations d'expressions

Nous présentons ici la représentation des expressions que nous avons adoptée. Elle permet de décrire des expressions contenant des variables scalaires, des tableaux ou des canaux de type entier ou structuré. Nous décrivons ensuite le mécanisme réalisant la collecte des valeurs des variables intervenant dans l'expression à évaluer lors de la propagation d'un homomorphisme sur DDD hiérarchique.

Nous avons représenté les expressions sous la forme d'arbres binaires, dont le nœud racine contient un opérateur, et son (ses) sous arbre(s) représentant la (les) sous-expression(s) (notées e_l et e_r) : $(\Box, e_l, e_r)$.

Les opérateurs $\Box$ correspondent à tous les opérateurs admissibles en Promela (excepté `run`). Les feuilles représentent :

- Les valeurs constantes (représentées sur un DDD représentant au moins une affectation et n'ayant qu'un seul chemin) : (d_{val})
- Les variables (var).

A cet ensemble s'ajoutent des opérateurs permettant de décrire des références, ce sont eux qui permettront d'identifier une variable, une structure, etc. :

- Les variables sont des références.
- La $p^{\text{ième}}$ case d'un tableau T de taille n est représentée par le nœud $([n], T, p)$. T doit être une référence et p peut être une expression quelconque. Si p ne contient pas de variable T, la *valeur* de $T[p]$ peut être obtenue par le produit orthogonal $([n], T, p) \star T = \sum_{i=0}^{n-1} T[i] \times (i = p)$.
- Le champ f d'une structure S est représenté par le nœud $(\bullet, S, f)$. S doit être une référence et f une *variable*.

Lors de la définition des homomorphismes faisant intervenir une expression, le test « $e \in expr$ » renvoyait une valeur booléenne indiquant si le nœud rencontré était étiqueté par une variable apparaissant dans $expr$. Dans l'approche hiérarchique, ce résultat n'est plus booléen et le test qui tient compte des quatre cas de figure est réalisé par une fonction membre de l'expression (noté $param()$) :

1. $expr.param(e) = \text{no}$: $expr$ n'est pas sensible à e. On propage le même homomorphisme.
2. $expr.param(e) = \text{var}$: e définit une *variable scalaire dans l'expression $expr$*. On remplace dans $expr$ toutes les variables e par chacune des valeurs de e, représentées sur les DDD renvoyés par l'algorithme Split. Ceci équivaut au terme $expr_{e:=x}$ utilisé jusqu'alors.

DDD « à plat »			
	$expr.param(e)$	$expr'$	$\langle\ \mathsf{hom}\ ()\ \rangle$
$d = e \xrightarrow{x} d'$	no	$expr$	$e \xrightarrow{x} \langle\ \mathsf{hom}\ (expr)\ \rangle$
	yes	$expr_{e:=x}$	$e \xrightarrow{x} \langle\ \mathsf{hom}\ (expr')\ \rangle$
DDD « hiérarchique »			
	$expr.param(e)$	$expr'$	$\langle\ \mathsf{hom}\ ()\ \rangle$
$d = e \xrightarrow{a_i} d_i$	no	$expr$	$e \xrightarrow{a_i} \langle\ \mathsf{hom}\ (expr)\ \rangle$
	var	$\left\{\begin{array}{c}\forall x \in Split(a_i),\\ expr_{(e)\leftarrow x}\end{array}\right\}$	$\left\{e \xrightarrow{x} \langle\ \mathsf{hom}\ (expr')\ \rangle\right\}$
	$struct$	$expr_{(.,(var),f)\leftarrow(f)}$	$\begin{array}{c}\langle\ \mathsf{rebuild}\ (e,1)\ \rangle\\ \circ\ \langle\ \mathsf{hom}\ (expr')\ \rangle\\ \circ\ \langle\ \mathsf{flat}\ ()\ \rangle\end{array}$
	$array(n)$	$\left\{\begin{array}{c}\forall i \in 0..n-1,\\ expr_{([],(e),p\star e)\leftarrow(i)}\end{array}\right\}$	$\left\{\begin{array}{c}\langle\ \mathsf{rebuild}\ (e,1)\ \rangle\\ \circ\ \langle\ \mathsf{hom}\ (expr')\ \rangle\\ \circ\ \langle\ \mathsf{selectExpr}\ (i = p \star e)\ \rangle\\ \circ\ \langle\ \mathsf{flat}\ ()\ \rangle\end{array}\right\}$

TAB. 7.1 Evaluation des homomorphismes définis sur des expressions hiérarchiques

3. $expr.param(e) = \text{struct}$: e est une structure dont on souhaite connaître la valeur du champ f. On supprime dans $expr$ la structure liée à e et on applique le nouvel homomorphisme composé à $\langle\ \mathsf{rebuild}\ ()\ \rangle$ à gauche et $\langle\ \mathsf{flat}\ ()\ \rangle$ à droite.
4. $expr.param(e) = \text{array(n)}$: e est un tableau de n cases dont on souhaite modifier ou lire la valeur rangée dans la case p (voir plus haut). Si l'on souhaite seulement lire la valeur en question, alors on remplace l'expression $([], e, p)$ par le produit orthogonal $([], e, p) \star e = \sum_{i=0}^{n-1} e[i] \times (i = p \star e)$. Par exemple, pour l'expression `T[T[p]]`, encodée sous la forme $(n, T, (n, T, p))$, le test $(T \in expr)$ renverra $array(n)$ et l'expression sera remplacée par $\sum_{i=0}^{n-1} T[i] \times (i = \sum_{j=0}^{n-1} T[i](j = n))$.

Le tableau 7.1 résume le mode de propagation d'un homomorphisme sensible à une expression $(expr)$: La première colonne donne la forme du DDD sur lequel s'applique l'homomorphisme. La deuxième colonne donne le résultat du test $expr.param(e)$. La troisième colonne donne l'expression (ou l'ensemble d'expressions) obtenues en modifiant $expr$ suivant les règles citées plus haut. La dernière colonne donne l'homomorphisme propagé (ou un ensemble d'homomorphismes si l'on a obtenu un ensemble d'expressions). Dans la partie supérieure (« DDD à plat ») sont repris les deux valeurs possibles du test $(e \in expr)$ avec les remplacements effectués (la variable e prend la valeur x) dans $expr$. Dans la partie inférieure (« DDD hiérarchique ») les quatres cas de figure décrits plus haut sont donnés.

Exemple 7.5 (Homomorphisme $\langle\ \mathsf{SelectExpr}\ ()\ \rangle$ en mode hiérarchique) *En reprenant les quatre* *cas de figure du tableau 7.1, on réécrit l'équation 5.4 de la proposition 5.4.1 :*

$$\langle\, \mathsf{selectExpr}\ (expr)\,\rangle(e, a_i) = \begin{cases} e \xrightarrow{x} \langle\, \mathsf{this}\,\rangle & \text{si } (e \in expr) = \text{no} \\ \sum_{x\in Split(e_i)} e \xrightarrow{a_i} \langle\, \mathsf{selectExpr}\ (expr_{e:=x})\,\rangle & \text{si } (e \in expr) = \text{var} \\ \begin{pmatrix} \langle\, \mathsf{rebuild}\ (e,1)\,\rangle \circ \\ \langle\, \mathsf{selectExpr}\ (expr_{(\bullet,e,f)\leftarrow f})\,\rangle \\ \circ \langle\, \mathsf{flat}\ ()\,\rangle(e, a_i) \end{pmatrix} & \text{si } (e \in expr) = \text{struct} \\ \begin{pmatrix} \langle\, \mathsf{rebuild}\ (e,1)\,\rangle \\ \circ \quad \sum_{i=1}^{n} \begin{pmatrix} \langle\, \mathsf{selectExpr}\ (expr_{([],e,p\star e)\leftarrow i} \wedge (p = i))\,\rangle \\ \circ \quad \langle\, \mathsf{flat}\ ()\,\rangle(e, e_i) \end{pmatrix} \end{pmatrix} & \text{si } (e \in expr) = \text{array} \end{cases}$$

Dans le cas de l'affectation d'une expression, l'homomorphisme est sensible à deux expressions : La première indiquant la référence de la variable affectée par la seconde expression. Le tableau 7.2 reprend le tableau 5.2 (évaluation inductive de l'homomorphisme $\langle\, \mathsf{setExpr}\ (v, expr)\,\rangle$) en distinguant tous les cas de figure possibles.

Les autres homomorphismes détaillés au chapitre 5 sont adaptés de façon identique en s'appuyant sur l'algorithme « Split » ainsi que sur $\langle\, \mathsf{flat}\ ()\,\rangle$ et $\langle\, \mathsf{rebuild}\ ()\,\rangle$ pour accéder au valeurs nécessaires à leur évaluation.

7.2.4 Parcours attelés hiérarchiques

Certains homomorphismes du chapitre 5 ne peuvent être adaptés aux DDD hiérarchiques à l'aide du seul tableau 7.1. Il s'agit des homomorphismes effectuant des parcours attelés (pour réaliser certains opérateurs pre). Ces homomorphismes sont paramétrés par un DDD sur lequel les opérateurs suivants pourront être appliqués.

- L'opérateur d'intersection $\cap$ qui est déjà décrit dans la section 6.3.
- L'opérateur de projection suivant *une valeur* : $(e, \alpha)|_x = \alpha(x)$.

Cet opérateur de projection peut être réalisé par l'homomorphisme $\langle\, \mathsf{project}\ ()\,\rangle$ qui s'applique sur les nœuds non terminaux :

$$\langle\, \mathsf{project}\ (v)\,\rangle(e, x) = \begin{cases} \langle\, \mathsf{id}\,\rangle & \text{si } x = v \\ 0 & \text{sinon} \end{cases}$$

On étend cet homomorphisme aux ensembles de valeurs à l'aide de la propriété suivante :

Définition 7.7 (Projection hiérarchique) *L'homomorphisme* $\langle\, \mathsf{set\_project}\ ()\,\rangle$ *de projection sur ensembles défini par*

$$\langle\, \mathsf{set\_project}\ (s)\,\rangle(e, a_i) = \begin{cases} \alpha_i & \text{si } a_i \cap s \neq \emptyset \\ 0 & \text{sinon} \end{cases}$$

$expr.param(e)$, $expr'$ / $v.param(e)$, v'	no, e	var, $\left\{ \begin{array}{c} \forall x \in Split(a_i), \\ (v) \leftarrow i \end{array} \right\}$	$struct$, $(., (v), f) \leftarrow (f)$	$array(n)$, $\sum_{i=0}^{n-1} T[i] \times (i = p \star e)$
no, v	$e \xrightarrow{e_i} \langle$ this $\rangle$	$e \xrightarrow{x}$ $\langle$ setExpr $(v, expr') \rangle$	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v, expr') \rangle$ $\circ$ $\langle$ flat $() \rangle$	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v, expr') \rangle$ $\circ$ $\langle$ flat $() \rangle$
var, $\left\{ \begin{array}{c} \forall x \in Split(a_i), \\ (v) \leftarrow i \end{array} \right\}$	$e \xrightarrow{x}$ $\langle$ downExpr $(v', expr) \rangle$	$e \xrightarrow{x}$ $\langle$ downExpr $(v', expr') \rangle$	Impossible	Impossible
$struct$ $(., (var), f) \leftarrow (f)$	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v', expr) \rangle$ $\circ$ $\langle$ flat $() \rangle$	Impossible	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v', expr') \rangle$ $\circ$ $\langle$ flat $() \rangle$	Impossible
$array(n)$, $\left\{ \begin{array}{c} \forall i \in 0 \ldots n-1, \\ ([], (e), p) \leftarrow (i) \end{array} \right\}$	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v', expr) \rangle$ $\circ$ $\langle$ selectExpr $(p \star e = i) \rangle$ $\circ$ $\langle$ flat $() \rangle$	Impossible	Impossible	$\langle$ rebuild $(e, 1) \rangle$ $\circ$ $\langle$ setExpr $(v', expr') \rangle$ $\circ$ $\langle$ selectExpr $(p \star e = i) \rangle$ $\circ$ $\langle$ flat $() \rangle$

TAB. 7.2 Evaluation inductive de $\langle$ setExpr $(v, expr) \rangle$ sur un couple (e, a_i), les expressions $expr'$ sont définies dans le tableau 7.1

Proposition 7.8 () *Soit v une valeur, on a* $\langle\ \mathsf{set\_project}\ (\{v\})\ \rangle = \langle\ \mathsf{project}\ (v)\ \rangle$ *et*

$$\sum_{x_i \in a_i} \langle\ \mathsf{project}\ (x_i)\ \rangle = \langle\ \mathsf{set\_project}\ a_i\ \rangle$$

Démonstration *: En annexe page 132.* □

Corollaire 7.9 *L'homomorphisme* $\langle\ \mathsf{set\_project}\ ()\ \rangle$ *est la réécriture de l'homomorphisme* $\langle\ \mathsf{project}\ ()\ \rangle$ *pour un étiquetage symbolique.*
Démonstration *: Elle est la conséquence directe de la proposition pour les homomorphismes inductifs sur espaces symboliques 6.5 et de la proposition 7.8.* □

Nous disposons donc de tous les outils nécessaires pour mettre en œuvre les méthodes développées chapitre 5 pour vérifier des systèmes Promela en exploitant la structure hiérarchique des programmes Promela.

7.3 Expérimentation

Dans cette section nous étendons les résultats du chapitre 5 en vérifiant les mêmes programmes avec un model-checker basé sur les DDD hiérarchiques lors de la vérification de propriétés CTL. La principale source de complexité apparaissant lors de la génération de l'espace des états accessibles. de ce fait, nous comparons les performances du nouveau prototype avec celui utilisant les DDD « plats » pour le calcul des états accessibles uniquement.

7.3.1 Saturation

Au cours de nos expérimentations le concept de saturation [CC04] s'est imposé comme étant une méthode viable pour générer rapidement l'espace des états accessibles avec une occupation en mémoire réduite.

Le principe de saturation adopté consiste à « regrouper » tous les opérateurs $post$ associés aux instructions d'un même processus et de les appliquer jusqu'à l'obtention d'un point fixe : il s'agit donc d'une recherche en largeur d'abord restreinte à un sous-ensemble du programme Promela. On réitère ce procédé pour *tous* les processus tant que de nouveaux états sont générés.

La saturation d'un processus peut s'exprimer localement par l'homomorphisme $\langle\ \mathsf{saturateProcess}\ ()\ \rangle$ défini comme suit :

Définition 7.10 (Saturation d'un processus) *Soit $post_p$ l'opérateur $post$ associé à toutes les instructions d'un processus p. La saturation du processus p peut être réalisée à l'aide de l'homomorphisme inductif suivant :*

$$\begin{aligned}
\langle\ \mathsf{saturateProcess}\ (p)\ \rangle(\top) &= \top \\
\langle\ \mathsf{saturateProcess}\ (p)\ \rangle(1) &= \top \\
\langle\ \mathsf{saturateProcess}\ (p)\ \rangle(d = (e, e_i)) &= \begin{cases} \langle\ \mathsf{saturate}\ (p)\ \rangle(d) & \text{si } e \in dependences(p) \\ e \xrightarrow{e_i} \langle\ \mathsf{this}\ \rangle & \text{sinon} \end{cases}
\end{aligned}$$

Où $\langle\ \mathsf{saturate}\ (p)\ \rangle$ *applique sur le DDD d l'opérateur $post_p$ jusqu'à l'obtention d'un point fixe.*

Le principe de saturation repose sur l'ordre d'application des opérateurs $post$ associés à toutes les instructions du programme Promela. Ces opérateurs étant regroupés par processus, il est toujours possible de modifier l'ordre de saturation des processus. De nombreuses stratégies de saturation ont été testées, la plus efficace (en terme d'occupation mémoire) est la plus simple, il s'agit de la saturation « Cyclique » : elle utilise un ordre circulaire de saturation (des strates du DDD principal les plus basses vers les strates les plus hautes ou inversement) et fait appel au ramasse-miettes après chaque saturation de processus.

Une autre stratégie de saturation, dite « Locale », se fait par l'emploi d'un *unique* homomorphisme $\langle$ saturateProcesses () $\rangle$ paramétré par un ensemble de processus (initialement l'ensemble des processus du programme). L'évaluation de ce processus sur un noeud étiqueté par la variable e est le suivant :

Algorithme 7.2 : Algorithme d'évaluation de l'homomorphisme $\langle$ saturateProcesses () $\rangle$

```
Input : Une variable e
Data : Un ensemble de processus à saturer processes
Data : Un ensemble de processus dépendants de e here ← ∅
Data : Un ensemble de processus indépendants de e down ← ∅
forall p ∈ processes do
    if e ∈ dependences(p) then
        here ← here ∪ p
    else
        down ← down ∪ p
while newstates do
    ⟨ saturateProcesses (down) ⟩
    forall p ∈ here do
        ⟨ saturateProcess (p) ⟩
```

A moins qu'il ne soit nécessaire d'activer des instructions de type `timeout`, l'ensemble des états accessibles est obtenu par une application unique de cet homomorphisme. Ce procédé s'est avéré remarquablement rapide mais le prix à payer est l'absence totale de contrôle quant à l'utilisation du ramasse-miettes. Ce mécanisme est donc plus gourmand en mémoire que les autres algorithmes de saturation, néanmoins, nous verrons qu'il est tellement rapide (même pour atteindre les limites de la mémoire utilisable), que nous préconisons son utilisation systématique, quitte à devoir appliquer une autre stratégie après occupation totale de l'espace mémoire.

7.3.2 Résultats

Ici, nous comparons les performances de nos deux prototypes basés respectivement sur la représentation d'espace d'états sur DDD « classiques » et sur des DDD hiérarchiques. Les deux représentations (hiérarchique et à plat) sont comparées sur une recherche en largeur d'abord. L'influence des méthodes de saturation est étudiée sur les seuls DDD hiérarchiques. Les performances sont données sur le tableau 7.3 (les calculs ont été effectués sur la même machine qu'au chapitre 5 à savoir une processeur Intel Pentium IV cadencé à 3,2 GHz), les valeurs les plus significatives sont données en caractères gras. Les durées sont exprimées en secondes (0 lorsque le temps écoulé est inférieur à une seconde) et la mémoire en mégaoctets. On reprend les données du tableau 5.3 dans les deux colonnes « à plat » que l'on compare avec les colonnes « hiérarchique ». Ces dernières exploitent trois modes de génération d'espace d'états :

– La recherche en largeur d'abord (colonnes « bfs »).
– La saturation basée sur une stratégie cyclique (colonnes « cyclique »).
– La saturation basée sur une stratégie locale (colonnes « locale »).

Nous ne nous intéressons qu'à la génération de l'ensemble des états accessibles, la vérification de propriétés CTL nécessitant des ressources matérielles moindres.

Impression générale

Au chapitre 5, nous avons souligné la supériorité des DDD pour l'analyse des systèmes présentant un fort degré de concurrence. Cette propriété est accentuée pour les DDD hiérarchiques lorsque le système comporte plus d'une dizaine de processus. Néanmoins cette plus-value n'est réellement tangible que lorsque les DAG sont correctement ordonnés, et aucune amélioration notable n'a été constatée pour l'analyse du protocole « Sliding Window », l'algorithme du boulanger et l'algorithme de Peterson. Les deux prochaines sections exploitent les résultats du tableau 7.3 pour détailler l'influence des méthodes mises en œuvre dans ce chapitre :

– L'analyse des performances de la saturation sur le dîner des philosophes et l'élection sur un anneau ajoute de nouveaux éléments pour le choix de l'ordre des variables sur un DDD.
– Les performances mitigées des trois autres systèmes (Sliding Window, Bakery et Peterson) soulignent les inconvénients liés aux homomorphismes $\langle \mathsf{flat}\ () \rangle$ et $\langle \mathsf{rebuild}\ () \rangle$.

Systèmes fortement concurrents : une question d'ordre

L'étude portant sur le dîner des philosophes et l'élection sur un anneau illustre la plus-value que procurent l'approche hiérarchique ainsi que les méthodes de saturation : Lorsqu'un ordre naturel peut être spécifié, les DDD hiérarchiques permettent de repousser les limites du model-checker décrit au chapitre 5 et les méthodes de saturation permettent d'achever le calcul de l'ensemble des états accessibles pour des systèmes comportant un millier de processus.

La possibilité de reporter les configurations de certaines variables sur des DDD partiels permet aussi de « clarifier l'allure » des différents DAG pour confirmer la qualité de l'ordre choisi : La figure 7.7 donne le premier niveau de hiérarchie de deux DDD représentant l'ensemble des états accessibles avec un ordre naturel ou un ordre quelconque donné par l'ordre d'apparition des éléments du programme Promela. Sur le DDD de gauche on distingue deux nattes régulières (le chemin tout à droite correspond à l'état qui précède l'initialisation des processus), tandis que sur le DDD de gauche, le partage des nœuds est très limité. De plus, comme l'ordre choisi « regroupe » les strates qui interagissent entre elles, les méthodes de saturation sont extrêmement efficaces (elles agissent sur un nombre très restreint de strates voisines) et le « cap » des 1000 philosophes est franchi dans des délais très courts.

L'observation la plus surprenante porte sur l'élection sur un anneau. Au chapitre 5 nous avons présenté ce système comme semblable à celui du dîner des philosophes mais avec une complexité intrinsèque des processus justifiant des performances moindres. On peut confirmer cette conjecture en observant les DDD de la figure 7.8 qui donne les DDD (ordonnés) représentant les ensembles d'états accessibles pour six philosophes (à gauche) et six électeurs (à droite). Seul le premier niveau de hiérarchie est donné. Pour le cas des électeurs, les DDD partiels sont bien plus complexes et

		No Order								Natural Order							
		à plat		Hiérarchique						à plat		Hiérarchique					
N	$\|S\|$	bfs		bfs		cyclic		local		bfs		bfs		cyclic		local	
		temps	mem	temps	mem	temps	mem	temps	mem	temps	mem	temps	mem	temps	mem	temps	mem
Dîner des Philosophes																	
5	244	0	4.3	0	2.1	0	1.8	0	2.7	0	5.5	0	2.1	0	1.7	0	1.8
10	5.9e4	31	17.6	19	25.0	13	10	10	56	9	11	4	5.7	0	1.8	0	2.2
13	1.6e06	1021	105	207	209	131	70	104	534	30	19.6	14	11	0	1.9	0	2.6
14	4.8e6	4534	203	448	434	278	139	238	1116	41	24.4	19	13	0	1.9	0	2.7
15	1.4e7	**18375**	**397**	**991**	**889**	**588**	**276**	*	*	54	29.6	25	16	0	1.9	0	2.8
20	3.5e9	*	*	*	*	*	*	*	*	182	64.1	80	35	1	2.2	0	3.5
50	7.2e23	*	*	*	*	*	*	*	*	**8006**	**994**	**3308**	**532**	5	4.7	1	8.3
100	5.2e47	*	*	*	*	*	*	*	*	*	*	*	*	22	13	3	22
1000	1.3e477	*	*	*	*	*	*	*	*	*	*	*	*	**9166**	**1078**	**258**	**1343**
Election sur un Anneau																	
5	5.7e3	8	6.3	11	5.6	2	3.0	1	7.2	11	6.2	6	3.9	0	2.3	0	3.7
10	3.2e7	4533	663	4553	849	174	58	87	349	**716**	**77**	**333**	**61**	5	3.98	2	14
15	1.9e11	*	*	*	*	8501	1821	*	*	*	*	2529	278	12	5.5	7	35
20	1.1e15	*	*	*	*	*	*	*	*	*	*	9891	771	23	6.8	14	65
30	3.6e22	*	*	*	*	*	*	*	*	*	*	*	*	55	10	34	151
100	1.5e75	*	*	*	*	*	*	*	*	*	*	*	*	671	32	**443**	**1866**
200	2.2e150	*	*	*	*	*	*	*	*	*	*	*	*	2770	63	*	*
1000	5.3e751	*	*	*	*	*	*	*	*	*	*	*	*	**73156**	**336**	*	*
Sliding Window																	
2	4.0e5	**486**	**17.8**	3447	27.5	1.6e5	284	**2244**	**774**	662	17	2741	29.2	2.2e5	267	**2569**	**657**
3	3.5e7	4.8e5	565	*	*	*	*	*	*	4.8e5	475	*	*	*	*	*	*
Peterson																	
2	208	0	3.6	0	1.7	0	1.7	0	1.9								
3	2.5e4	7	5.9	53	5.4	25	6.0	5	7.5								
4	6.4e6	**1278**	**113**	31042	214	27160	201	4454	396								
Bakery																	
4	1.8e5	17	8.9	702	13	806	18	167	42								
5	1.6e7	379	46	148017	142	110241	163	14767	491								
6	2.2e9	**61901**	**455**	*	*	*	*	*	*								

TAB. 7.3 Performances sur DDD Classiques et Hiérarchiques

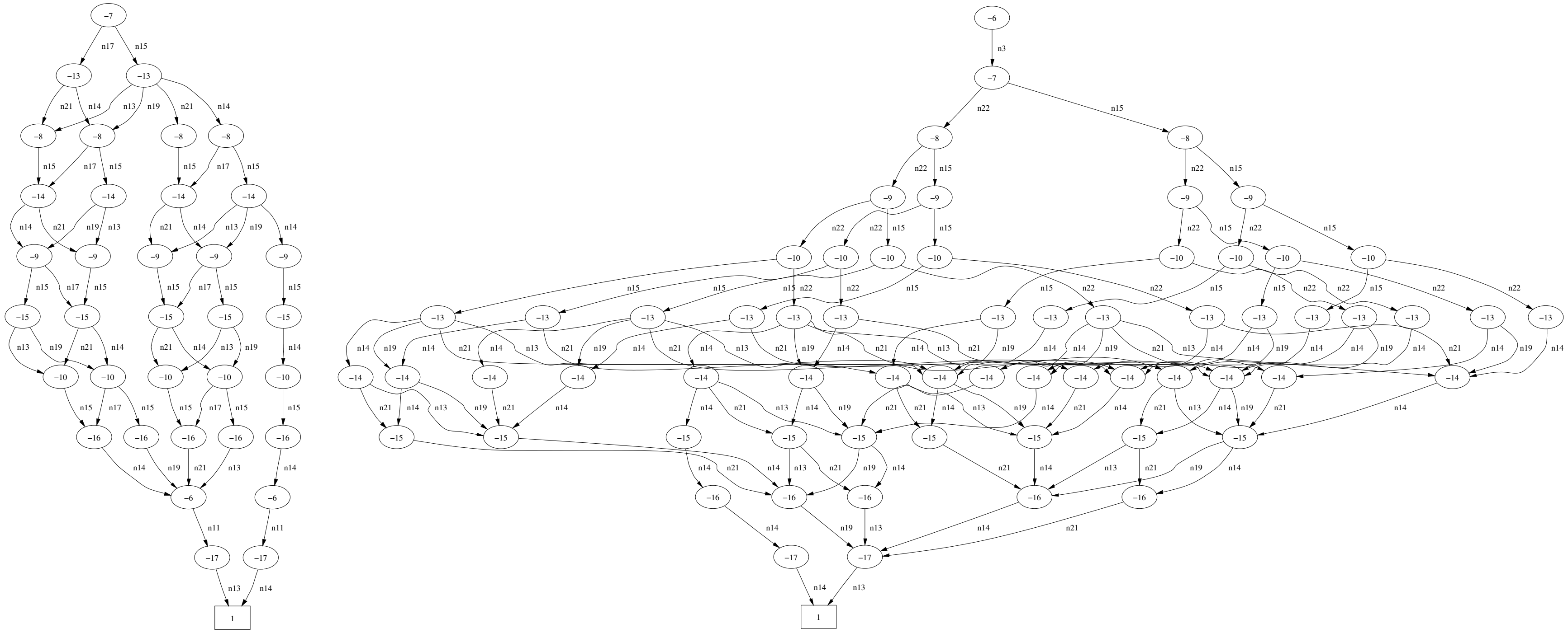


FIG. 7.7 SDD obtenu pour 4 philosophes, avec un ordre naturel ou non

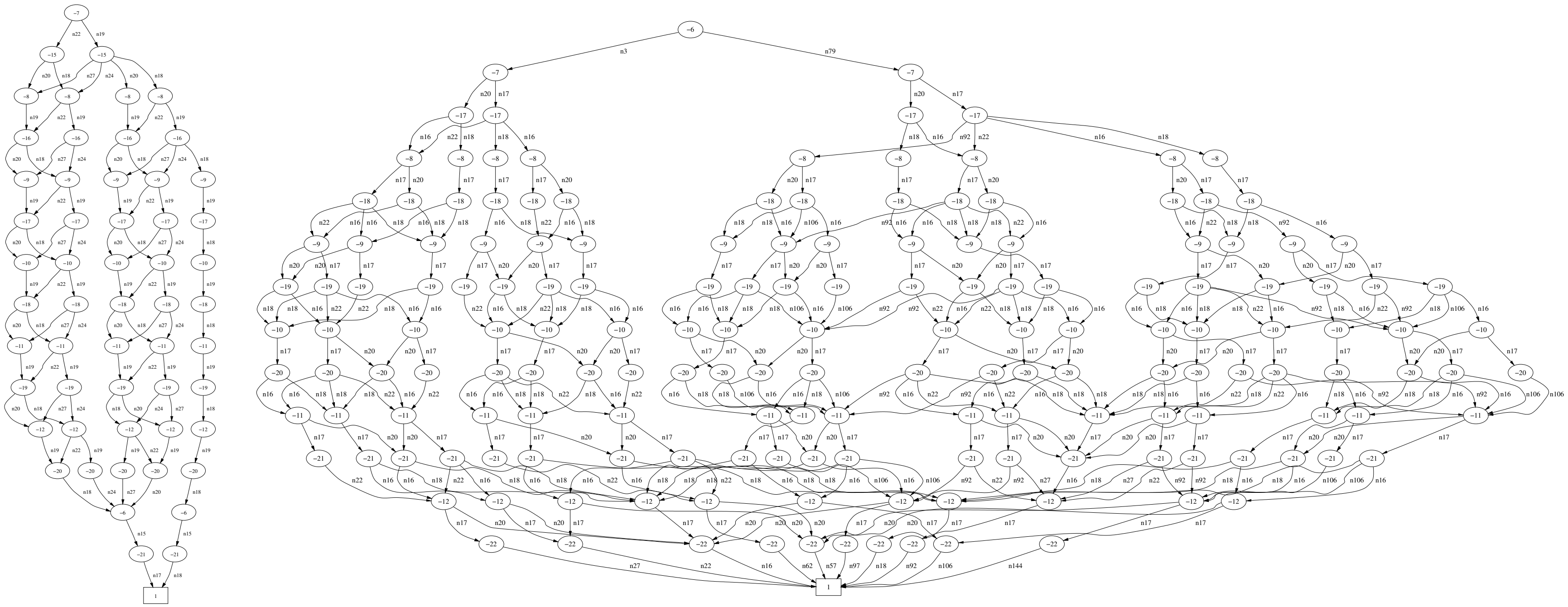


FIG. 7.8 DAG (ordonnés) générés pour 6 philosophes ou 6 électeurs sur un anneau

plus nombreux que ceux apparaissant pour les philosophes : les différentes configurations initiales de l'élection sur un anneau produit un Diagramme sans aucune régularité en comparaison avec le premier.

Or, lorsqu'on applique une saturation cyclique, notre prototype est non seulement capable d'appréhender autant d'électeurs que de philosophes (jusqu'à un millier de processus), mais l'occupation en mémoire est sensiblement moindre. On explique cette propriété par la différence de comportement entre les cuillers (des philosophes) et les jetons des électeurs : Contrairement aux cuillers, les jetons « circulent » en sens unique, et il en va de même pour les données propagées par les homomorphismes.

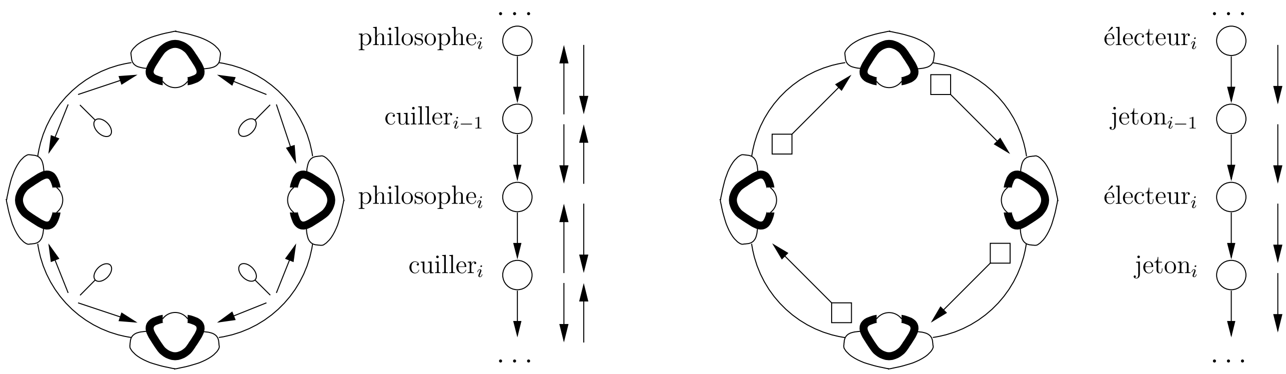


FIG. 7.9 « Circulation » des données dans le cas des philosophes et de l'élection sur un anneau

La figure 7.9 résume cette différence, sur la partie gauche est schématisé le problème des philosophes, les cuillers peuvent « passer » vers le philosophe situé immédiatement à droite ou à gauche. Sur la partie droite est schématisé le problème de l'élection. Les jetons circulent en sens unique. Sur les chemins du DDD cette différence a un poids : la donnée indiquant la position de la cuiller peut « remonter » sur le chemin, ce qui implique l'application de l'homomorphisme $\langle$ up () $\rangle$ tandis que dans le cas de l'élection, les jetons ne font que « descendre » de strate en strate. En utilisant une stratégie de circulation cyclique de la racine vers le nœud terminal, chaque processus est saturé dans le contexte où il est à même de « produire » le plus de nouveaux états et l'homomorphisme $\langle$ up () $\rangle$ n'est utilisé que pour la saturation du premier processus. La contre-performance (en terme de temps) de la saturation locale confirme cette observation : la saturation des processus se faisant des strates les plus basses vers les strates les plus hautes, les contextes d'exécution sont moins propices à la génération de nouveaux états.

Ce dernier point soulève à nouveau l'importante question : « qu'est ce qu'un bon ordre ? » Un nouvel élément vient s'ajouter à la topologie du système pour répondre à cette question : s'il est possible de distinguer un « flux » de circulation des données, l'ordre d'apparition des variables et des processus sur le DAG devra suivre ce flux. Dans le cas qui nous préoccupe, ceci a permis de limiter l'utilisation de l'homomorphisme $\langle$ up () $\rangle$ (seulement entre le premier et le dernier processus de l'ordre) alors qu'elle est inévitable pour le problème des philosophes. Il en va de même pour l'ordre de saturation des processus : les homomorphismes se propageant de la racine vers les nœuds terminaux du DAG leurs évaluations conduiront à une génération accrue de nouveaux états.

Les systèmes faiblement concurrents : les limites de $\langle$ flat () $\rangle$ et $\langle$ rebuild () $\rangle$

Pour le prototocole Sliding Window et les algorithmes du boulanger et de Peterson, la hiérarchie et les méthodes de saturation apparaissent comme des facteurs contre-productifs. La forte dépendance entre les différents processus explique l'inefficacité de la saturation : les processus sont tous conditionnés par un unique élément du programme :

– des tableaux de valeurs partagés pour les deux algorithmes,
– la clause `timeout` dans le protocole Sliding Window,

qui atténuent le degré de parallélisme des systèmes.

L'autre contre-performance soulignée par l'étude de ces trois systèmes concerne l'apport de la hiérarchie qui est manifestement négatif. Avec les DDD « à plat » il n'est pas possible d'appréhender ce type de systèmes avec plus de six processus, et la présence de certaines instructions critiques, sensibles à la majorité des éléments du système, accroît le nombre de strates mises à plat (à l'aide de $\langle$ flat () $\rangle$). Dans de tels cas de figure, les DAG issus des calculs intermédiaires présentent un nombre beaucoup trop important de strates aplanies, et ce sont cette fois les nœuds hiérarchiques qui apparaissent comme surnuméraires.

7.4 Conclusion

Nous avons montré comment adapter les méthodes du chapitre 5 à l'étiquetage symbolique des arcs qu'implémentent les SDD. Le cadre théorique fourni dans le chapitre 4 étant toujours valable, nous avons pu implémenter un nouveau prototype de model-checker symbolique, basé sur les SDD, tirant parti de la hiérarchie des programmes Promela et des méthodes de saturation. Couplés aux méthodes de saturation, les SDD apportent une plus-value considérable pour des systèmes fortement concurrents tels que le dîner des philosophes ou l'élection sur un anneau. Néanmoins, l'utilisation des SDD suppose la mise en œuvre de mécanismes coûteux en espace mémoire pour des systèmes défavorables aux DDD. Ces systèmes restent des problèmes académiques, très éloignés de ceux qui sont généralement écrits en langage DPN, Ce sont plutôt des systèmes fournis par le concepteur d'applications industrielles réellement implantées qu'il nous faut tester.

Troisième partie

Application - Conclusion et Perspectives

Chapitre 8

Exemples Réalistes

Les Diagrammes de Décision d'Ensembles couplés aux méthodes de saturation repoussent les limites de l'application des techniques symboliques de model-checking. La comparaison d'outils sur des systèmes « académiques » permet de se faire une idée sur les classes de systèmes pour lesquels cette approche est la plus adaptée. Le but de ce chapitre est d'éprouver les techniques que nous avons mises en œuvre sur des exemples plus réalistes, à savoir des systèmes réellement implantés et dont le modèle a été fourni par le concepteur.

Sommaire

8.1 Deux protocoles

8.1.1 ZCSP

Présentation

Le protocole ZCSP (Zero Copy Secured Protocol [GDDW02]) est un protocole pour les machines parallèles procurant à un processus un accès direct aux zones mémoires d'autres processus implantés sur des machines distinctes. Le principe d'un protocole "zéro-copie" est de court-circuiter les couches système dans lesquelles plusieurs recopies dans les zones mémoire sont effectuées. Il s'agit d'un protocole extrêmement robuste palliant les défaillances du réseau : Pour un message donné, l'émetteur envoie une série de paquets indexés suivant un ordre croissant. Le récepteur n'acquitte un message que s'il a bien reçu tous les paquets suivant cet ordre et si aucun d'eux n'est invalide. Dans notre modèle, nous avons simulé les pertes (coupures ou brouillage) par des pertes franches, les messages invalides étant ignorés. A chaque émission est associé un délai de garde qui borne le temps nécessaire estimé pour la transmission du message suivie de la transmission de l'acquittement. Tout message n'ayant pas été acquitté à temps sera émis une nouvelle fois *in extenso* et l'acquittement doit être celui du *nouveau* message expédié. Enfin, un message peut être émis avant l'acquittement d'un message qui a été envoyé antérieurement ce qui suppose de tenir à jour une table des messages en attente d'acquittement. La figure 8.1 donne trois exemples de communication.

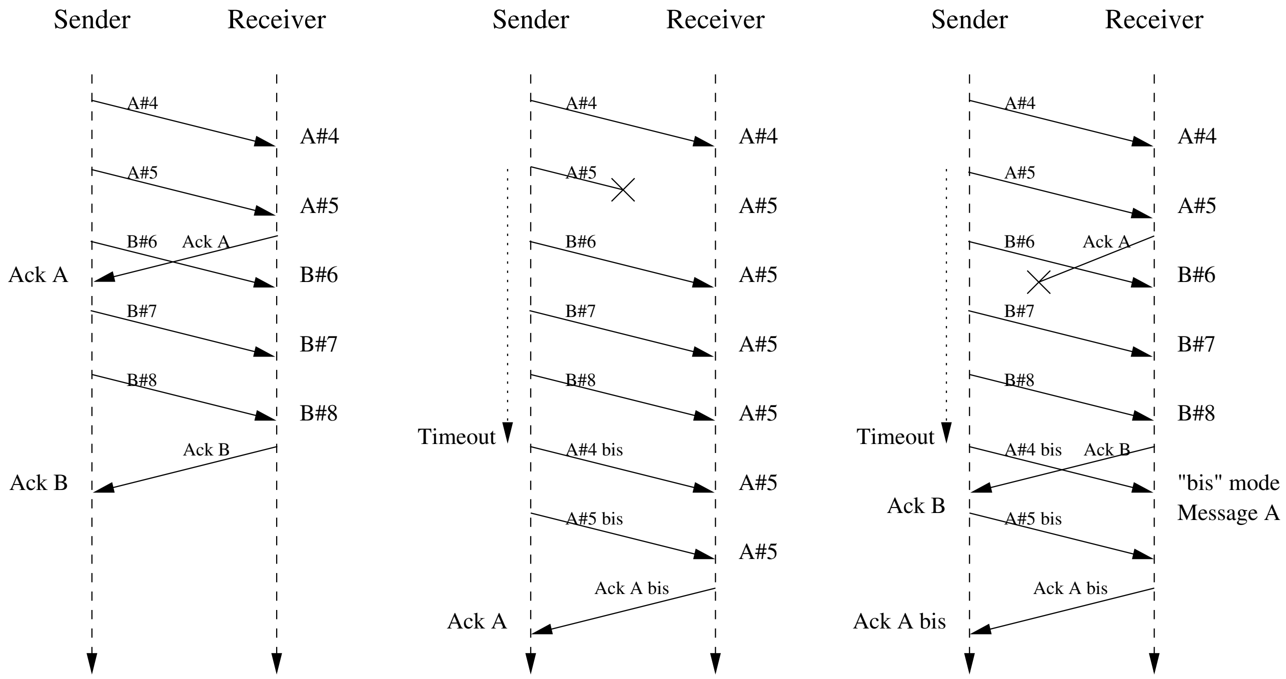


FIG. 8.1 Exemple de communication utilisant le protocole ZCSP

- Dans le diagramme de gauche, la communication s'effectue sans perte de paquet ou d'acquittement. On compte deux messages : A, dont les paquets sont indexés par 4 et 5, et B dont les paquets sont indexés par 6, 7 et 8. Le récepteur reçoit bien les numéros de paquet attendus, et les acquittements sont bien transférés.
- Dans le diagramme du milieu, un paquet est perdu, le récepteur rejette tous les autres paquets jusqu'à réception d'un nouveau paquet portant le bon index (5). Celui-ci se trouve dans message étiqueté « bis » réémis lorsque le délai donné pour la réception d'un acquittement est dépassé.
- Dans le diagramme de droite, A et B sont correctement émis, mais l'acquittement de A est perdu. Après réception *complête* du message « A bis », le récepteur émet un nouvel acquittement qui sera cette fois bien transmis. Dans ce scénario, le message B est acquitté avant le message « A ».

Le modèle Promela du protocole ZCSP est décrit sur la figure 8.2. Suivant l'état de la table des messages en attente, l'émetteur envoie des paquets vers le récepteur qui transmet les acquittements vers un processus chargé de mettre à jour la table. L'exécution concurrente des processus d'émission (« Sender ») et de mise à jour (« Update ») simule le comportement d'un émetteur tel qu'il est implanté dans le circuit. Les accès à la table se font par l'intermédiaire des variables Eld_P, FF, et FT

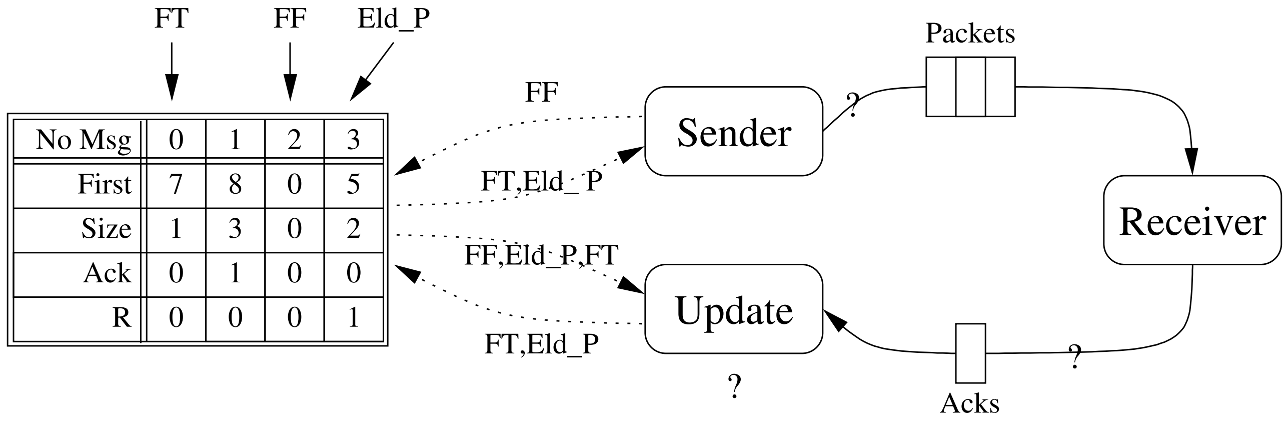


FIG. 8.2 Fonctionnement global du protocole ZCSP

qui désignent respectivement le message le plus ancien, la première case vide du tableau (à accès circulaire) et le premier message susceptible de ne pas être acquitté à temps. Les points d'interrogation indiquent les zones du système pour lesquelles à été introduit de l'indéterminisme pour simuler les pertes de paquets ou l'arrivée tardive d'un acquittement.

Caractéristiques du systèmes

Le protocole ZCSP présente les caractéristiques suivantes :

1. il comporte un faible nombre (3) de processus différents,
2. les processus sont fortement séquentiels et de complexité très importante,
3. deux d'entre-eux se partagent des variables partagées, dont la table des messages en attente qui est la principale source de complexité du système.

Le protocole ZCSP appartient donc à la catégorie de systèmes pour lesquels le model-checker SPIN est le mieux adapté.

Propriétés vérifiées

Le propriétés vérifiées ont été construites comme suit :

1. Le processus « Update » pouvant suivre des cycles qui ne provoquent aucune évolution du système, ou considérer *toute* réception d'un acquittement comme ayant dépassé le delai de garde, nous avons écarté ces scenarii en ajoutant deux conditions d'équité :
   - Le processus emetteur est activé infiniment souvent : $fair_s = \mathbf{GF}sender\_activated$.
   - Toutes les réceptions d'acquittement ne peuvent être considérées comme tardives : $fair_a = \mathbf{GF}ack$.
2. La propriété de vivacité exprime qu'un nouveau message est envoyé infiniment souvent $nms = \mathbf{GF}new\_message\_sent$ *si les conditions d'équité sont remplies* : $f_l = (fair_s \wedge fair_a) \implies nms$.
3. La propriété de terminaison exprime que tous les messages en attente seront acquittés ($empty\_table$) dans le futur si *aucun nouveau message n'est émis* et si les conditions d'équité sont remplies : $f_t = (\neg nms \wedge f_s \wedge f_a) \implies \mathbf{F}empty\_table$.

Résultats

La vérification des propriétés de vivacité et de terminaison avec SPIN a été réalisée au prix d'une simplification extrême du système [BED03]. *Les méthodes symbolique n'ont pu construire l'ensemble des états accessibles.* Ce résultat corrobore les observations faites aux chapitres 5 et 7 selon lesquelles SPIN est plus performant que nos prototypes pour des systèmes séquentiels.

8.1.2 ANI

Présentation

Le système ANI est un modèle initialement décrit en DPN. Il s'agit d'un réseau d'interconnexion en anneau permettant plusieurs communications simultanées.

Le système se compose de tâches (Maîtresses ou Esclaves) communiquant par l'intermédiaire de Wrapper (Maîtres ou Esclaves) disposés sur l'anneau qui est divisé en tronçons : Chaque tronçon est délimité par deux registres « Pipe » et contient un jeton qui est le support d'une communication. En un cycle, le jeton transite d'un Pipe vers le suivant en véhiculant une information au travers des wrappers connectés sur ce tronçon.

Les modèles des différentes plateformes ANI ont été conçus par [Tak03] dans l'optique de vérifier des propriétés LTL avec SPIN.

La figure 8.3 donne la version la plus simple possible : le système ne comporte qu'une seule instance de chaque type de composant (Un couple Tâche-Wrapper Maître, un autre couple Esclave et un Pipe). Une seconde plateforme a été proposée, doublant le nombre de chaque élément (voir figure

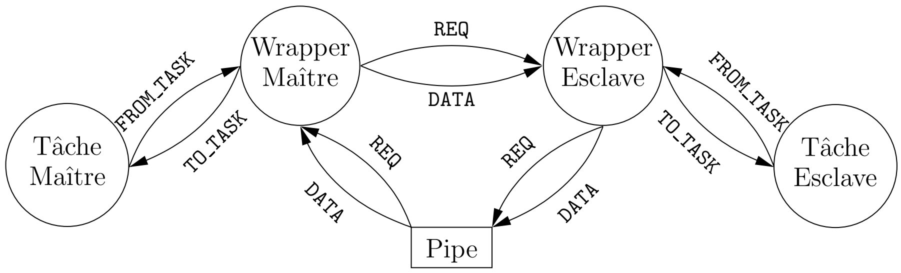


FIG. 8.3 La première plateforme ANI

8.4)

Ces systèmes se distinguent de ceux pourvus d'une topologie en anneau, abordés aux chapitres 5 et 7, sur trois points :

- Certains processus (les tâches) sont externes à l'anneau.
- Les éléments du système sont hétérogènes.
- Les informations transitant sur l'anneau sont transférées à l'aide de deux canaux distincts, l'un acheminant les données tandis que l'autre achemine les requêtes.

A ces différences s'ajoute le rôle des pipes qui agissent comme des barrières de synchronisation, régulant la progression des différentes requêtes.

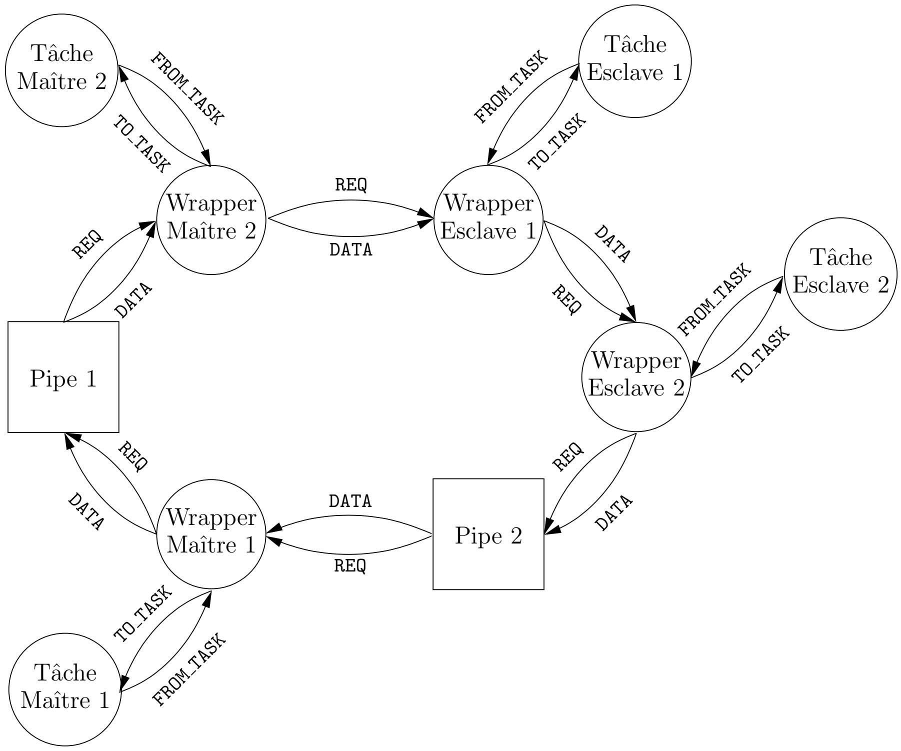


FIG. 8.4 La seconde plateforme ANI

SPIN		Hierarchical DDD						Ordered Hierarchical DDD					
		bfs		cyclic		local		bfs		cyclic		local	
time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
2.5	11	7796	143	3746	141	875	863	4709	107	2626	50	435	155

TAB. 8.1 Performances comparées de SPIN et de notre outil pour la première plate-forme ANI

Résultats

Les performances obtenues pour la construction de l'ensemble des états accessibles sont données sur le tableau 8.1. Les temps sont donnés en secondes et l'occupation de l'espace mémoire en méga-octets. Seule la première plate-forme a pu être analysée. La topologie ne permet pas de construire un bon ordre pour les DDD, ni d'exploiter la localité des transitions.

8.2 Le réseau SPIN

8.2.1 Présentation

Le réseau SPIN [ACG+03] (Scalable, Programmable, Integrated Network) est une architecture de communication pour systèmes intégrés sur puce. Il se présente sous la forme d'un réseau étagé de routeurs décrit sur le figure 8.5. Au bas de cette figure sont représentés les abonnés au réseau, il s'agit d'initiateurs et de cibles VCI. Suivant l'adresse de destination d'un paquet reçu par un routeur, la retransmission se fait :

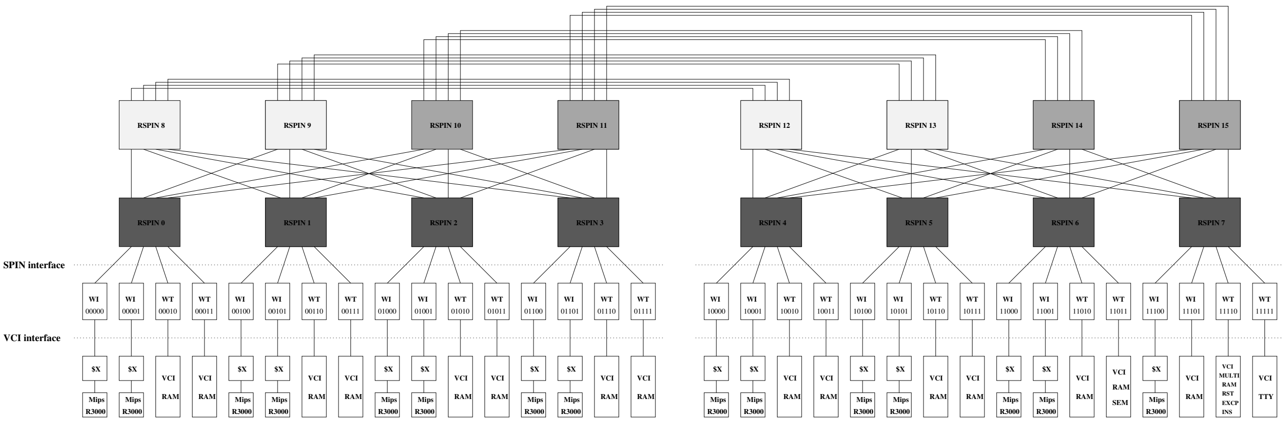


FIG. 8.5 Un réseau SPIN

– soit de façon déterministe vers l'étage inférieur,
– soit de façon indéterministe vers l'étage supérieur (si la cible se situe dans un autre sous-réseau dont le routeur n'est pas l'ascendant suivant la topologie arborescente).

Le routage d'un paquet (*i.e.* d'un ensemble de plusieurs mots) se fait suivant la politique Wormhole : Tous les mots se suivent sur un même chemin sans entrelacement de messages ni bufferisation complète d'un message avant l'envoi vers un autre routeur.

Tout mot reçu au niveau du destinataire VCI (cible) est acquitté par un mot réponse circulant vers l'initiateur. Ce procédé d'acquittement est source d'interblocages, comme celà a pu être démontré « à la main » ou à l'aide de l'outil SPIN.

La solution proposée pour éviter ces interblocages est de spécialiser les routeurs du plus haut niveau à l'acheminement d'un type unique de mot (requête ou réponse). On parle alors de réseau *séparé*. Dans la figure 8.5, les routeurs grisés du niveau le plus haut sont ceux qui sont consacrés à l'acheminement des réponses, les routeurs blancs n'acheminant *que* des requêtes.

Afin de s'assurer que le réseau séparé est exempt d'interblocage, il a été proposé d'employer l'outil SPIN pour vérifier des propriétés de vivacité et de terminaison sur des réseaux d'interconnection exploitant cette topologie. Dans la suite, nous décrivons le modèle Promela décrit par le concepteur de l'architecture, les vérifications faites avec SPIN puis celles faites avec notre outil.

8.2.2 Modèles Dégradés à deux et quatre routeurs

Les modèles dégradés à deux et quatre routeurs du réseau SPIN sont donnés au centre et à droite de la figure 8.6. Nous détaillerons plus tard la décomposition d'un routeur donnée sur la partie gauche de la figure. Dans cette section nous détaillons tous les éléments du système, tels qu'ils ont été fournis par le concepteur.

Données

Le réseau SPIN achemine des mots « requête » des initiateurs vers les cibles et autant de mots « réponse » des cibles vers les initiateurs. Ici, nous nous focalisons sur le transit de ces mots, aussi, la validité des données acheminées ne nous concerne pas, et seul l'en-tête des mots est représenté. Il se compose de quatre champs entiers :

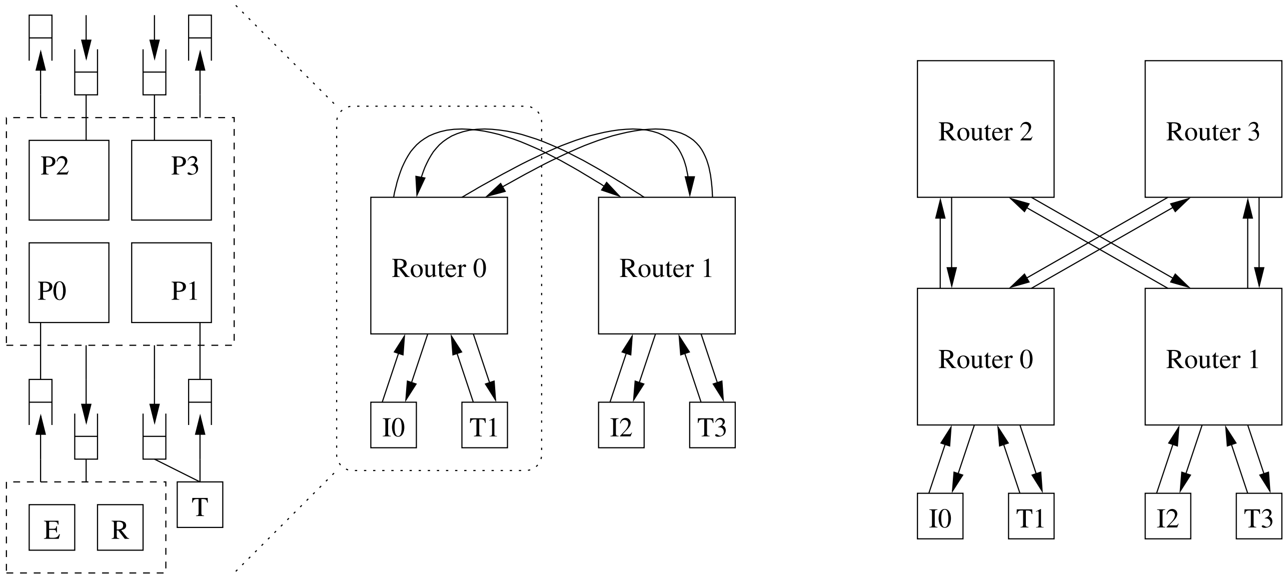


FIG. 8.6 Modèles dégradés à deux et quatre routeurs du réseau SPIN

1. L'adresse de l'abonné émetteur du mot (ici 0, 1, 2, ou 3).
2. L'adresse du destinataire du mot.
3. Le type de mot (requête ou réponse).
4. Une signature indiquant s'il s'agit du premier, du dernier ou d'un mot intermédiaire d'un paquet.

Ces informations transitent à travers différents éléments mémorisants du réseau que nous décrivons dans le paragraphe suivant.

Processus et Canaux

Sur la partie gauche de la figure 8.6, nous donnons le mode d'implémentation des éléments du réseau :

1. Les routeurs sont décomposés en quatre ports distincts (P0, P1, P2 et P3) afin de modéliser la possibilité qu'offre un routeur d'acheminer jusqu'à quatre paquets d'origines différentes simultanément.
2. Les initiateurs sont décomposés en deux processus distincts E et R respectivement chargés de l'émission de requêtes et de la réception des réponses. Cette séparation permet à l'initiateur de recevoir un paquet « réponse » associé au paquet « requête » en cours d'émission. Lorsqu'un paquet est intégralement acquitté (tous les mots « réponse » ont été reçus), le processus R accorde au processus E le droit d'envoyer un nouveau paquet, par l'intermédiaire d'une communication bufferisée.
3. Les cibles ont un comportement déterministe ; la réception d'un mot « requête » par une cible s'accompagne de l'émission d'un mot « réponse » pour lequel la signature est conservée et les adresses d'émission et de réception sont permutées.

A chaque port d'entrée d'un processus est associé un canal de communication pouvant contenir un mot. Les canaux acceptent plusieurs écrivains. Pour assurer le routage suivant la politique Wormhole, il faut garantir l'exclusivité en écriture d'un processus, pour un paquet donné, durant *toute* l'émission

Paquets/Messages	transitions	temps (s)	mémoire (Mb)
2	2.3e8	35381	94
3	1.9e8	25018	138
4	2.3e8	37335	96
5	1.9e8	25023	139
6	2.3e8	36000	99
7	5.6e7	11263	5.9

TAB. 8.2 Performances du model-checker SPIN sur le réseau SPIN

d'un paquet. Elle est réalisée par l'utilisation d'un unique jeton par canal, pris par le processus avant l'émission du premier mot d'un paquet, et rendu après l'émission du dernier.

8.2.3 Configurations

La comparaison des différentes approches de model-checking a été effectuée sur différentes configurations des modèles à deux et quatre routeurs en tenant compte des paramètres suivants :

1. **Mode Séparé :** Lors de l'envoi d'un paquet vers un étage supérieur, le choix du routeur destinataire devient déterministe suivant la nature (requête ou réponse) du paquet transféré.
2. **Mode Synchrone :** Pour réduire le nombre de chemins dans l'arbre d'exécution, on utilise la clause `timeout` pour simuler un mécanisme de synchronisation des processus.
3. **Configurations d'émission :** Pour les réseaux à deux et quatre routeurs, nous avons traité sept configurations d'émission :
   - Six configurations pour lesquelles l'initiateur 0 (resp.2) émet systématiquement vers la cible 3 (resp. 1) des messages de taille fixe (2, 3... ou 7 paquets).
   - Une configuration pour laquelle l'adresse de la cible et la taille du message peuvent varier après chaque émission.

8.2.4 Résultats

Dans cette section nous donnons les performances de l'outil SPIN ainsi que celles de notre outil sur les différents modèles détaillés dans la section précédente.

Vérifications accomplies avec l'outil SPIN

Nous avons vérifié la propriété de vivacité **GF**$progress$ (où $progress$ désigne l'envoi d'une *nouvelle* requête par l'initiateur 0), sur les 6 modèles synchrones séparés pour lesquels la configuration d'émission est figée. Les performances sont données dans le tableau 8.2. Si l'on relâche la contrainte de synchronisation, l'outil SPIN n'est plus en mesure d'achever les calculs avant l'occupation totale de l'espace mémoire (2Go). Il en va de même pour une recherche des états accessibles dans le cas du mode non séparé.

Les modèles implémentant toutes les tailles de messages avec adressage aléatoire n'ont pu être vérifiées avec SPIN. De plus, SPIN n'a pas pu achever la construction de l'espace des états accessibles ou vérifier la propriété de vivacité sur les modèles à quatre routeurs séparés. En outre, même les

			mode séparé							mode non séparé				
			SPIN		DDD					DDD				
					Cyclic		Local				Cyclic		Local	
	n	$\|S\|$	temps	mem	temps	mem	temps	mem	$\|S\|$	temps	mem	temps	mem
SYNCH	2	4.5e8	698	8.3	88	27	18	80	1.2e9	484	140	111	131
	3	4.5e8	481	9.1	134	32	22	100	1.2e9	854	207	149	577
	4	4.5e8	508	8.9	191	36	26	114	1.2e9	1293	250	189	743
	5	4.6e8	486	9.3	259	41	29	126	1.2e9	1785	274	210	813
	6	4.6e8	512	9.0	340	49	33	139	1.2e9	2397	327	232	901
	7	4.6e8	491	9.3	420	53	**36**	**152**	1.2e9	3017	342	245	945
ASYNCH	2	1.9e8			74	13	12	45	8.1e8	502	52	55	179
	3	2.2e9			197	28	27	95	9.4e9	1345	119	121	383
	4	1.7e10			402	51	50	170	7.2e10	2734	222	223	700
	5	8.3e10			664	75	73	241	3.4e11	4431	335	323	973
	6	2.7e11			957	102	95	303	1.1e12	6176	418	414	1219
	7	6.5e11			1258	120	120	390	2.6e12	**7881**	**492**	**519**	**1562**

TAB. 8.3 Résultats comparatifs pour des réseaux SPIN 2 routeurs (configuration d'émission figée)

interblocages n'ont pu être décelés alors que la recherche de contre-exemple constitue pourtant une force de l'outil SPIN.

Vérification accomplies avec les DDD hiérarchiques

Cette section porte sur la construction de l'ensemble des états accessibles et la recherche de deadlocks. Le résultat des études comparatives sur un réseau à deux routeurs est résumé sur le tableau 8.3. Les temps sont donnés en secondes et l'occupation de l'espace mémoire en mégaoctets. Les configurations d'émissions sont figées et les modes synchrones, asynchrones, séparés ou non séparés sont tous pris en compte.

Pour les seules configurations pour lesquelles l'outil SPIN parvient à construire l'ensemble des états accessibles, les performances en terme d'occupation mémoire sont en faveur de l'outil SPIN. Néanmoins, l'occupation mémoire de notre outil reste raisonnable même pour une machine d'entrée de gamme (152Mo) et les durées de calcul sont considérablement moins importantes : il faut moins de temps pour achever les calculs pour les six configurations d'émissions qu'il n'en faut pour l'outil SPIN pour calculer l'ensemble des états accessibles d'une seule configuration d'émission.

La construction de l'ensemble des états accessibles pour les autres modes (séparé ou non, asynchrone ou non) a été rendue possible avec notre outil. Avec une machine pourvue de 2Go de mémoire vive, la saturation « locale » permet d'achever l'ensemble des calculs en moins d'une heure de temps. La saturation « cyclique » est une bonne alternative si l'on dispose d'une mémoire vivre moindre (512 Mo) mais il faudra patienter une demi-journée.

Si l'on compare les performances de notre méthode suivant les différents cas de figure, on constate que le passage d'un modèle vers un autre modèle plus « général » (passage au mode non séparé ou au mode asynchrone) s'accompagne d'un accroîssement des ressources matérielles nécessaires. Il en va de même lorsque l'on construit l'ensemble des états accessibles, pour des réseaux à deux ou quatre routeurs réalisant toutes les configurations d'émission possibles, que seul notre outil a pu construire (hormis le réseau à quatre routeurs non séparé asynchrone). Les performances sur des

	n	mode séparé $\|S\|$	temps	mem	mode non séparé $\|S\|$	temps	mem
SYNCH	2	4.3e9	225	21	9.7e9	803	188
	4	2.4e13	469	27	5.0e13	1715	407
ASYNCH	2	1.7e12	5791	429	7.1e12	31033	1884
	4	1.4e14	11549	1025	*	*	*

TAB. 8.4 Recherche complêtes des états accessibles sur les Réseaux SPIN à 2 ou 4 routeurs

DDD hiérarchiques avec saturation cyclique sont données dans le tableau 8.4. Les temps sont donnés en secondes et l'occupation de l'espace mémoire en mégaoctets.

Ajoutons à celà que l'approche utilisant des DDD « classiques » n'avait produit aucun résultat sur toutes les configurations que nous avons testées dans cette section. En effet, en raison de nombre d'informations contenues dans chaque élément mémorisant, le DDD représentant un réseau à deux routeurs atteint une profondeur supérieure à 200 nœuds. Cette profondeur est réduite à une cinquantaine de nœuds lorsque l'on utilise la hiérarchie suggérée par le modèle Promela, ce qui explique les différences d'efficacité entre les deux représentations.

A partir de l'ensemble des états accessibles, nous avons alors pu vérifier des propriétés de sûreté comme l'absence d'états n'ayant pas de successeur indiquant un blocage complet du réseau (clause `timeout`), cette recherche ne nécessite aucun espace mémoire supplémentaire et est effectuée en un temps négligeable par rapport à la recherche des états accessibles.

Apparitions de Deadlocks

La recherche d'états n'ayant pas d'état successeur ne permet pas d'identifier avec précision des interbocages confinés dans une partie du réseau et permettant un progression normale du reste du réseau (livelock). Néanmoins, comme les interblocages sont dûs à l'acheminement des requêtes, une telle situation présentera au moins une requête bloquée. Comme le model-checker explore toutes les possibilités d'exécution, et en raison de la politique de routage Wormhole, *il doit apparaître dans l'ensemble des états accessibles* une configuration pour laquelle la cible ayant émis la requête en question reste bloquée (il suffit pour cela que le nombre de paquets « réponse » dépasse le nombre d'éléments mémorisants qui apparaissent sur le chemin emprunté par le message jusqu'à la zone de blocage). A partir de cette configuration, le model checker construit tous les chemin d'exécution possibles, y compris ceux où toutes les nouvelles requêtes sont destinées à la cible bloquée. Or, ces derniers scénarii mènent à un blocage *total* du système (deadlock). En d'autres termes, la possibilité de livelock implique l'apparition de deadlock. Par contraposée, l'absence de deadlock prouve qu'il ne peut y avoir de livelock.

Conclusion sur le réseau SPIN

Les modèles dégradés du réseau SPIN font intervenir un grand nombre de processus de moyenne et de forte complexité. Malgré le comportement séquentiel et complexe des ports modélisés, notre outil est considérablement plus efficace que l'outil SPIN. Le model-checking symbolique basé sur les DDD hiérarchiques et utilisant des méthodes de saturation est donc une alternative sérieuse pour la vérification de systèmes pourvus d'un fort degré de parallélisme.

8.3 Conclusion

Nous avons expérimenté notre approche sur des systèmes issus d'applications réelles, de complexité moyenne. Au cours de ces expérimentations, nous avons tenu à ne pas toucher à la modélisation, ce que l'on fait généralement à des fins d'optimisation pour la vérification.

L'analyse des systèmes ZCSP et ANI corrobore les résultats des chapitres 5 et 7 quant à l'inefficacité des méthodes symboliques utilisant les DDD sur les sytèmes faiblement concurrents.

Néanmoins la vérification sur le réseau SPIN, à deux et à quatre routeurs, montre que ces méthodes apportent une réelle plus-value pour la vérification de systèmes très concurrents.

Chapitre 9

Conclusion

L'objectif principal de cette thèse est le développement d'une méthode de vérification automatique de systèmes

- qui représentent des réseaux de processus concurrents asynchrones,
- dont les communications sont assurées par des canaux adoptant la politique FIFO,
- pour lesquels l'implémentation (matérielle ou logicielle) n'est pas définie *a priori*.

Les outils existant pour la vérification de tels systèmes font défaut. Le choix d'un sous-ensemble de Promela nous a permis de conserver un lien sémantique étroit avec les applications définies au sein de la plateforme de conception matérielle/logicielle Disydent, tout en procurant un cadre de comparaison avec d'autres techniques de model-checking.

Nous proposons l'utilisation des méthodes de model-checking symbolique basé sur les Diagrammes de Décision de Données (DDD, [CEPA$^+$02]), et les méthodes inductives qui leurs sont associées, pour les raisons suivantes :

- les méthodes symboliques sont plus efficaces que les méthodes énumératives, comme celles proposées par le model-checker SPIN [Hol97b],
- les DDD s'affranchissent d'une représentation *au bit près* des données ce qui les rend plus adaptés à la vérification de programmes Promela.

Les contributions formelles apportées par cette thèse sont les suivantes :

1. La définition d'un cadre de construction de DDD ordonnés, compatible avec les instanciations multiples de variables, qui assure les propriétés de canonicité et de bonne définition. Des règles de construction des homomorphismes inductifs ont été énoncées pour garantir la préservation de l'ordre. Un mécanisme de replacement des variables ($\langle \mathsf{replace}() \rangle$, inspiré de [CEPA$^+$02], a été défini pour réordonnancer les variables après application d'un homomorphisme brisant l'ordre initial.
2. La démonstration que les Diagrammes de Décision d'Ensembles (SDD, [TM04, CTM05]) *sont* des Diagrammes de Décision de Données pourvus d'une implémentation différente. Ce résultat permet de conserver le cadre théorique lié aux DDD tout en tirant parti de l'étiquetage symbolique des arcs introduit par les SDD.

Ces contributions confirment les intuitions formulées par [CEPA$^+$02] pour les DDD et [TM04, CTM05] pour les SDD.

Les contributions techniques apportées concernent le développement de deux prototypes de model-checker symbolique. Les points suivants ont été traités :

1. La définition de l'ensemble des traitements élémentaires, nécessaires pour le model-checking symbolique, associés à la sémantique du langage Promela et indépendants du type des données appréhendées.
2. La définition des homomorphismes réalisant ces traitements élémentaires.
3. La définition des modes d'accès, en lecture et en écriture, à des variables appartenant à différents niveaux de hiérarchie dans les DDD hiérarchiques.
4. La mise à l'épreuve des prototypes sur des jeux de tests académiques et des systèmes réalistes ainsi que des comparaisons avec des outils et techniques standard (SPIN pour les techniques énumératives et les BDD pour les techniques symboliques).

Les études réalisées montrent que le model-checking symbolique basé sur les DDD est une alternative sérieuse pour la vérification de systèmes fortement concurrents, mais à délaisser dans le cas de systèmes séquentiels. Pour les systèmes présentant une hiérarchie exploitable, les DDD hiérarchiques, couplés aux méthodes de saturation, apportent une plus-value significative.

Problèmes subsistants

D'une manière générale, l'expérience accumulée lors de l'élaboration des deux prototypes peut contribuer de manière significative à l'amélioration de la bibliothèque [DDD] et des techniques pour la vérification d'applications DPN :

1. Au chapitre 5 nous avons évoqué la construction d'homomorphismes réalisant en un seul parcours la compositions de plusieurs homomorphismes élémentaires. Leur construction est simple mais leur généralisation conduit à une explosion du nombre d'homomorphismes à réaliser. Nous suggérons l'emploi d'homomorphismes localement composables (voir annexe E) pour résoudre ce problème. Bien que les techniques de saturation réduisent l'utilité de tels homomorphismes, il reste quelques cas critiques (en particulier lorsque des strates distantes sont concernées) où leur emploi serait profitable.
2. Au chapitre 7 nous avons mis en œuvre des méthodes (Split, $\langle$ flat () $\rangle$ et $\langle$ rebuild () $\rangle$) pour adapter les homomorphismes du chapitre 5 à l'étiquetage symbolique :
   - L'algorithme de différenciation (Split) peut être appelé plusieurs fois sur des mêmes DDD et lorsqu'un ensemble de valeurs (pour une variable donnée) a besoin d'être différencié il le demeurera durant toute l'exécution du model-checker. Il est donc souhaitable de conserver ces valeurs énumérées à l'aide d'un cache, lequel devrait s'enrichir à la suite de d'application d'opérations ensemblistes.
   - Les mécanismes de mise à plat d'un niveau de hiérarchie pourraient être émulés à l'aide d'homomorphismes dits « multilinéaires » qui, appliqués sur les DDD partiels, renverraient les DDD propagés en aval du délimiteur de fin de bloc (EOB).

   Ces deux perspectives de recherche sont développées dans l'annexe E.
3. Le cache de l'espace de travail, censé éviter les redondances de calcul est remis en cause par les méthodes de saturations. Le compromis temps-mémoire penche désormais en faveur du temps de calcul, et le mode de génération de nouveaux nœuds suggère d'abandonner (du moins, en partie) le cache, quitte à ralentir le calcul pour économiser de l'espace mémoire.
4. Les DDD sont sensibles à l'ordonnancement des variables : quelques critères intuitifs ont été énoncés et vérifiés en pratique. Il serait profitable d'approfondir cette étude et de définir précisément des critères de « bon ordre » pour les DDD.

5. L'ajout d'une composante hiérarchique dans les DDD était à l'origine une alternative pour exploiter les symétries de certains réseaux de Petri [CTM05]. Certaines symétries analogues se retrouvent dans les systèmes que nous étudions mais il est impossible d'en spécifier la nature dans un programme Promela. Or, notre objectif premier était la vérification de systèmes décrits en DPN et non en Promela : il est tout à fait envisageable d'élaborer un langage accessible aux concepteurs, traduisible en DPN et plus adapté à la vérification exploitant les DDD hiérarchiques. L'idée sous-jacente est la définition de *processus hiérarchiques* pour maximiser l'exploitation de la modularité et des hiérarchies d'un système. L'annexe E donne un exemple d'exploitation des symétries proposé par [CTM05] transposable au réseau d'interconnexion SPIN.

Quatrième partie

Annexes

Annexe A

Saturation

Cette annexe donne un exemple justifiant les méthodes de saturations pour la construction des états accessibles par rapports aux méthodes traditionnelles de recherche en profondeur ou en largeur d'abord.

Exemple A.1 (DFS, BFS et Saturation) *La figure A.1 montre le fonctionnement des trois paradigmes de génération d'espace d'états que l'on compare pour la création de couples* $(x, y) \in \{0 \dots n\}^2$ *aléatoires en partant de* $(0, 0)$ *et en incrémentant l'un ou l'autre membre du couple de façon concurrente. La partie supérieure présente l'ensemble des points du plan discret à construire. L'ensemble des états générés à chaque étape de calcul est entouré par des pointillés. La partie inférieure de la figure donne les MDD associés à l'ensemble des états nouvellement atteints : On suppose ici que l'espace d'états générés est représenté par un arbre de la forme* $x \xrightarrow{val_x} y \xrightarrow{val_y} 1$ *et que l'opérateur post s'écrit* $post_x \cup post_y = ((x \leftarrow x + 1) \cup (y \leftarrow y + 1))$.

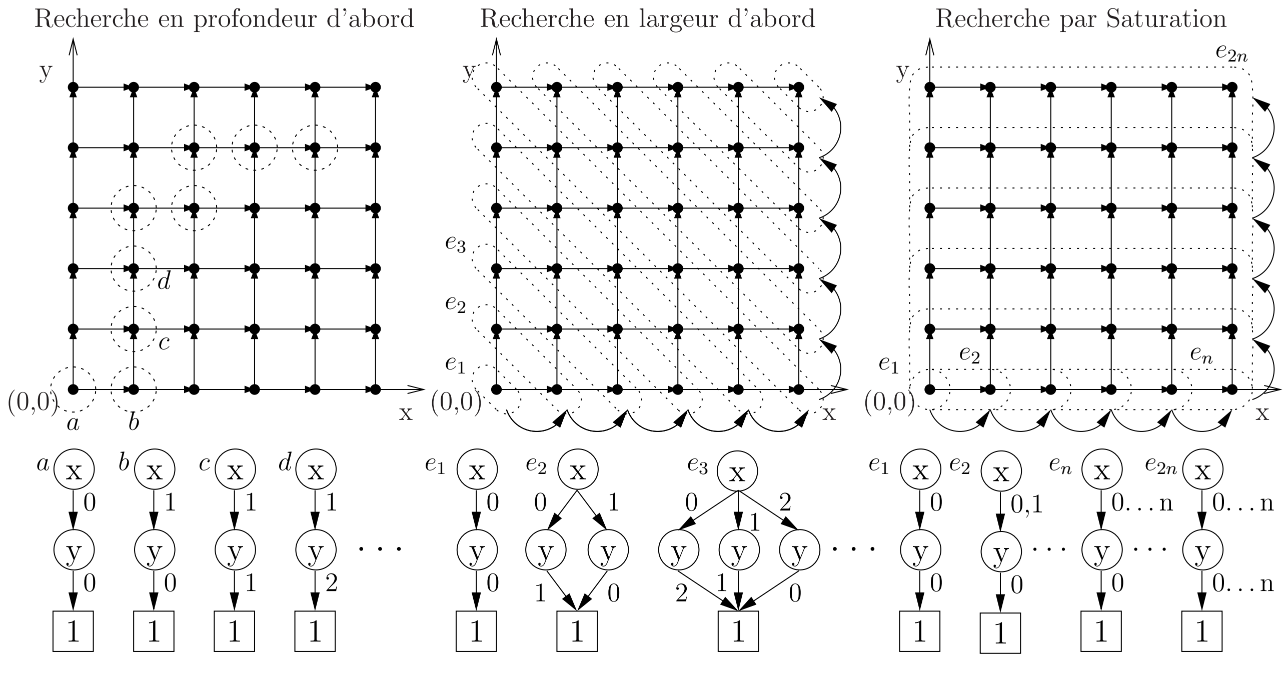


FIG. A.1 Génération aléatoire de points du plan suivant trois modes de recherche différents

- *Dans le schéma de gauche (en profondeur d'abord), chaque flèche correspond à la création d'un point à partir d'un autre. Les points sont générés un par un, il existe plusieurs chemins menant au même point. Le nombre d'applications de l'opérateur* $post$ *suit* $O(n^2)$.
- *Dans le schéma du millieu (en largeur d'abord) on génère les points par ensemble, l'incrémentation de l'abscisse ou de l'ordonnée se fait indifféremment, par la création de l'ensemble des points* $x + y = c, c \in \{0 \ldots 2 \times n\}$. *Le nombre d'application de l'opérateur* $post$ *suit* $O(n)$.
- *Dans le schéma de droite (par saturation), on génère d'abord tous les points d'ordonnée nulle avant d'incrémenter l'ordonnée. La complexité du calcul reste à* $O(n)$, *mais on a différencié l'incrémentation de l'abscisse et celle de l'ordonnée, ce qui revient d'abord à évaluer les points* $x = c, c \in \{0 \ldots n\}$ *sur une unique strate* x, *puis les ensembles* $y = c, c \in \{0 \ldots n\}$ *à partir des précédents. Le nombre d'application de l'opérateur* $post$ *suit toujours* $O(n)$.

Le nombre d'applications de l'opérateur $post$ *est identique dans les deux derniers cas, et pourtant la saturation est beaucoup plus efficace avec cette représentation d'espace d'états : dans le cas d'une recherche en largeur d'abord, les calculs font intervenir deux transitions différentes sur deux strates différentes, l'arbre des nouveaux états générés à l'étape* p *s'écrit* $\bigcup_{i=0}^{p} x \xrightarrow{i} y \xrightarrow{p-i} 1$ *qui est un ensemble de chemins disjoints, alors que dans le cas de la saturation, on applique une seule transition (*$post_x$*) autant de fois que nécessaire sur une même strate (x) avant de passer à l'autre transition (*$post_y$*) sur la strate correspondante (*y*), ainsi, l'arbre des états générés à l'étape* $p \leq n$ *est* $x \xrightarrow{\{0 \ldots p\}} y \xrightarrow{0} 1$ *et pour* $n < p \leq 2 \times n$ *l'arbre devient* $x \xrightarrow{\{0 \ldots n\}} y \xrightarrow{\{0 \ldots p-n\}} 1$. *Le nombre de nœuds intermédiaires passe donc de* $O(n^2)$ *à* $O(n)$ *si l'on applique la saturation à la place de la recherche en largeur d'abord.*

Annexe B

Preuves des Propositions

B.1 Chapitre 4

Proposition 4.8 (Ordres et Concaténation) *Soient deux DDD d_1 et d_2 d'ordres respectifs $\mathcal{O}_1$ et $\mathcal{O}_2$, et bien définis, alors $d = d_1 \cdot d_2$ est bien défini et d'ordre $\mathcal{O} = \mathcal{O}_1 \cdot \mathcal{O}_2$.*
***Démonstration** : Elle se fait par récurrence :*

- *si $d_1 \equiv 0$ $d_1 \cdot d_2 = 0$ qui est bien défini et compatible avec n'importe quel ordre.*
- *si $d_1 = 1$, $d_1 \cdot d_2 = d_2$ qui est bien défini respecte l'ordre $\emptyset \cdot \mathcal{O}_2 = \mathcal{O}_2$*
- *si $d_1 = (e, \alpha)$, alors $\mathcal{O}_1 = (e, \mathcal{O}'_1)$. $d = d_1 \cdot d_2 = (e, \sum_{x \in (Dom(e) \cup \sharp)} \left(e \xrightarrow{x} \alpha(x) \cdot d_2\right)$, d respecte l'ordre $\mathcal{O}_1 \cdot \mathcal{O}_2$ si*
  - *$e \in S$, et si $\forall x \in Dom(e)$, $\alpha(x) \cdot d_2$ est bien défini et respecte l'ordre $\mathcal{O}'_1 \cdot \mathcal{O}_2$ ou si*
  - *$e \in C$, et si $\forall x \in Dom(e)$, $\alpha(x) \cdot d_2$ est bien défini respecte l'ordre $\mathcal{O}_1 \cdot \mathcal{O}_2$ et $\alpha(\sharp)$ est bien défini et respecte l'ordre $\mathcal{O}'_1 \cdot \mathcal{O}_2$.*

On remarque que les deux dernières assertions sont vérifiées par la définition 4.6 si $\mathcal{O}' = (\mathcal{O}'_1 \cdot \mathcal{O}_2)$. La propriété étant vérifiée au niveau des feuilles (1 ou 0), elle l'est par récurrence sur chacun des nœuds du résultat $d_1 \cdot d_2$. □

Proposition 4.13 () *Soit s une variable scalaire ($s \in S$) et $d = (v, \alpha)$ un DDD bien défini d'ordre $(v, (s, \mathcal{O})) = (v) \cdot (s) \cdot \mathcal{O}$, alors $d' = \langle\ \mathsf{replace}\ (s)\ \rangle(d)$ est bien défini et d'ordre $(s, (v, \mathcal{O})) = (s) \cdot (v) \cdot \mathcal{O}$.*
***Démonstration** : Elle se fait par récurrence :*

- *Si $v \in S$ alors $d = \sum_{x \in Dom(v)} v \xrightarrow{x} d_x$ où d_x respecte l'ordre $(s) \cdot \mathcal{O}$, et $d' = \sum_{x \in Dom(v)} \langle\ \mathsf{up}\ (v, x)\ \rangle d_x$ qui est bien défini et respecte l'ordre $(s) \cdot (v) \cdot \mathcal{O}$ (proposition 4.11).*
- *Si $v \in C$ alors on raisonne par récurrence, d peut s'écrire $d_v \cdot d_s$ où d_v encode toutes les affectations de v et d_s respecte l'ordre $(s) \cdot \mathcal{O}$ et en vertu de la définition 4.2 d_v peut s'écrire*

$$d_v = \sum_{v_i \in Dom(v)} \bullet_{j=1}^{n} \left(v \xrightarrow{v_{ij}} 1\right) \cdot v \xrightarrow{\sharp} 1$$

il vient :

$$\begin{aligned} d' &= \langle\ \mathsf{replace}\ s\ \rangle(d) \\ &= \langle\ \mathsf{replace}\ s\ \rangle \left(\sum_{v_i \in Dom(v)} \bullet_{j=1}^{n} \left(v \xrightarrow{v_{ij}} 1 \right) \cdot v \xrightarrow{\sharp} 1 \right) \cdot d_s \\ &= \sum_{v_i \in Dom(v)} \left(\bigcirc_{j=1}^{n} (\langle\ \mathsf{up}\ (v, v_{ij})\ \rangle) \circ \langle\ \mathsf{up}\ (v, \sharp)\ \rangle \right) \circ \langle\ \mathsf{replace}\ s\ \rangle(d_s) \end{aligned}$$

$\langle\ \mathsf{replace}\ s\ \rangle(d_s)$ *respectant l'ordre* $(s)\cdot\mathcal{O}$*, en appliquant récursivement la propriété 4.11, on démontre que* d' *est bien défini et respecte l'ordre* $(s)\cdot(v)\cdot\mathcal{O} = (s,(v,\mathcal{O}))$*.* □

B.2 Chapitre 5

Proposition 5.3 (Homomorphisme $\langle\ \mathsf{downExpr}\ ()\ \rangle$) *L'homomorphisme* $\langle\ \mathsf{downExpr}\ (var, expr)\ \rangle$ *défini par*

$$\begin{aligned} \langle\ \mathsf{downExpr}\ (s, expr)\ \rangle(1) &= \top \\ \langle\ \mathsf{downExpr}\ (s, expr = cst)\ \rangle(e) &= var \xrightarrow{cst} e \xrightarrow{x} \langle\ \mathsf{id}\ \rangle \\ \langle\ \mathsf{downExpr}\ (s, expr \neq cst)\ \rangle(e,x) &= \begin{cases} \langle\ \mathsf{up}\ (e,x)\ \rangle \circ \langle\ \mathsf{downExpr}\ (s, expr_{e:=x})\ \rangle & \text{si } e \in expr \\ \langle\ \mathsf{up}\ (e,x)\ \rangle \circ \langle\ \mathsf{downExpr}\ (s, expr)\ \rangle & \text{sinon} \end{cases} \end{aligned}$$

Réalise l'homomorphisme $\langle\ \mathsf{replace}\ (s)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle$*.*
Démonstration *: Elle est immédiate si* $expr = cst$ *et lorsque l'homomorphisme est appliqué sur* 1*. Pour un couple* (e,α) *la démonstration se fait par récurrence :*

– $e \in expr$*,*

$$\begin{aligned} & \langle\ \mathsf{replace}\ (s)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle(e,\alpha) \\ =\ & \langle\ \mathsf{replace}\ (s)\ \rangle \circ \sum_{x \in Dom(e)} \left(e \xrightarrow{x} \langle\ \mathsf{collectExpr}\ (s, expr_{e:=x})\ \rangle \right) \\ =\ & \sum_{x \in Dom(e)} (\langle\ \mathsf{up}\ (e,x)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr_{e:=x})\ \rangle) \end{aligned}$$

– $e \notin expr$*,*

$$\begin{aligned} & \langle\ \mathsf{replace}\ (s)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle(e,\alpha) \\ =\ & \langle\ \mathsf{replace}\ (s)\ \rangle \circ \sum_{x \in Dom(e)} \left(e \xrightarrow{x} \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle \right) \\ =\ & \sum_{x \in Dom(e)} (\langle\ \mathsf{up}\ (e,x)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle) \end{aligned}$$

Ainsi, $\langle\ \mathsf{downExpr}\ (var, expr)\ \rangle(e,\alpha) = \langle\ \mathsf{replace}\ (s)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle(e,\alpha)$ *si* $\forall x \in Dom(e)$*,* $\langle\ \mathsf{downExpr}\ (var, expr)\ \rangle(\alpha(x)) = \langle\ \mathsf{replace}\ (s)\ \rangle \circ \langle\ \mathsf{collectExpr}\ (s, expr)\ \rangle(\alpha(x))$ *comme la propriété est vraie pour* $expr = cst$ *et* $d = 1$ *elle est vérifiée par récurrence pour tout couple* (e,α)*.* □

B.3 Chapitre 6

Proposition 6.4 (Evaluation symbolique des opétateurs ensemblistes) *Soient deux DDD,* $d_1 = (e_1, \alpha_1)$ *et* $d_2 = (e_2, \alpha_2)$ *tels que* $e_1 = e_2$*,* $\alpha_1(x) \sum_{i=1}^{m} d_{1i}\chi_{1i}(x)$ *et* $\alpha_2(x) \sum_{j=1}^{n} d_{2j}\chi_{2j}(x)$*. Les opérateurs d'union, d'intersection et de différence entre* d_1 *et* d_2 *peuvent être symboliquement évalués à l'aide des équations suivantes :*

$$
\begin{aligned}
(\alpha_1 + \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{pmatrix} & (d_{1i} + d_{2j}) & (\chi_{1i}(x) \times \chi_{2j}(x)) \\ + & d_{1i} & (\chi_{1i}(x) - \chi_{2j}(x)) \\ + & d_{2j} & (\chi_{2j}(x) - \chi_{1i}(x)) \end{pmatrix} \\
(\alpha_1 \times \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} (d_{1i} \times d_{2j})(\chi_{1i}(x)\chi_{2j}(x)) \\
(\alpha_1 \backslash \alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{pmatrix} & (d_{1i} \backslash d_{2j}) & (\chi_{1i}(x) \times \chi_{2j}(x)) \\ + & d_{1i} & (\chi_{1i}(x) - \chi_{2j}(x)) \end{pmatrix}
\end{aligned}
$$

Démonstration *:*

$$
\begin{aligned}
(\alpha_1 \diamond \alpha_2) &= \sum_{i=1}^{m} d_{1i}\chi_{1i} \diamond \sum_{j=1}^{n} d_{2j}\chi_{2j} \\
&= \sum_{i=1}^{m} d_{1i}\chi_{1i}\underbrace{(\sum_{j=1}^{n}(\chi_{2j} + 1 - \chi_{2j}))}_{=1} \diamond \sum_{j=1}^{n} d_{2j}\chi_{2j}\underbrace{(\sum_{i=1}^{m}(\chi_{1i} + 1 - \chi_{1i}))}_{=1} \\
&= \sum_{i=1}^{m}\sum_{j=1}^{n}(d_{1i}\chi_{1i}\chi_{2j} + d_{1i}\chi_{1i}(1 - \chi_{2j})) \diamond \sum_{i=1}^{m}\sum_{j=1}^{n}(d_{2j}\chi_{2j}\chi_{1i} + d_{2j}\chi_{2j}(1 - \chi_{1i})) \\
&= \sum_{i=1}^{m}\sum_{j=1}^{n}(d_{1i}\chi_{1i}\chi_{2j} + d_{1i}(\chi_{1i} - \chi_{2j})) \diamond \sum_{i=1}^{m}\sum_{j=1}^{n}(d_{2j}\chi_{2j}\chi_{1i} + d_{2j}(\chi_{2j} - \chi_{1i})) \\
&= \left(\sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}\chi_{1i}\chi_{2j} + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}(\chi_{1i} - \chi_{2j})\right) \\
&\quad \diamond \left(\sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}\chi_{2j}\chi_{1i} + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}(\chi_{2j} - \chi_{1i})\right)
\end{aligned}
$$

Les quatres fonctions qui interviennent dans cette dernière équation ont pour support $\chi_1\chi_2$*,* $\chi_1 - \chi_2$ *ou* $\chi_2 - \chi_1$*, dont les intersections sont vides deux à deux. Si l'on remplace* $\diamond$ *par* $+$*,* $\times$ *ou* $\backslash$*, en* *appliquant les rêgles de commutativité (pour $+$ et $\backslash$) ou de distributivité (pour $\times$), on obtient :*

$$\begin{aligned}
(\alpha_1+\alpha_2) &= \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}\chi_{1i}\chi_{2j} + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}\chi_{2j}\chi_{1i} \\
&\quad + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}(\chi_{1i}-\chi_{2j}) + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}(\chi_{2j}-\chi_{1i}) \\
(\alpha_1\times\alpha_2) &= \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}\chi_{1i}\chi_{2j} \times \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}\chi_{2j}\chi_{1i} \\
(\alpha_1\backslash\alpha_2) &= \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}\chi_{1i}\chi_{2j} \backslash \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}\chi_{2j}\chi_{1i} \\
&\quad + \sum_{i=1}^{m}\sum_{j=1}^{n} d_{1i}(\chi_{1i}-\chi_{2j}) \backslash \sum_{i=1}^{m}\sum_{j=1}^{n} d_{2j}(\chi_{2j}-\chi_{1i})
\end{aligned}$$

En réunissant les termes sous les mêmes sommes, on obtient les propriétés suivantes :

$$\begin{aligned}
(\alpha_1+\alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{pmatrix} & (\alpha_{1i}+\alpha_{2j}) & (\chi_{1i}(x)\times\chi_{2j}(x)) \\ + & \alpha_{1i} & (\chi_{1i}(x)-\chi_{2j}(x)) \\ + & \alpha_{2j} & (\chi_{2j}(x)-\chi_{1i}(x)) \end{pmatrix} \\
(\alpha_1\times\alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} (\alpha_{1i}\times\alpha_{2j})(\chi_{1i}(x)\chi_{2j}(x)) \\
(\alpha_1\backslash\alpha_2)(x) &= \sum_{i=1}^{m}\sum_{j=1}^{n} \begin{pmatrix} & (\alpha_{1i}\backslash\alpha_{2j}) & (\chi_{1i}(x)\times\chi_{2j}(x)) \\ + & \alpha_{1i} & (\chi_{1i}(x)-\chi_{2j}(x)) \end{pmatrix}
\end{aligned}$$

$\square$

Proposition 6.5 (Homomorphismes inductifs sur espaces symboliques) *Soient c un SDD et $\phi(e,x)_{e\in E, x\in Dom(e)}$ une famille d'homomorphismes.*

$$\Phi(d) = \begin{cases} 0 & \text{si } d=0 \\ \top & \text{si } d=\top \\ c & \text{si } d=1 \\ \sum_{i=1}^{n}\Phi(e,a_i)(d_i\chi_i(x)) & \text{si } \begin{cases} d &= (e,\alpha) \\ \alpha(x) &= \sum_{i=0}^{n}(d_i\chi_i(x)) \\ \Phi(e,a_i) &= \sum_{x\in a_i}\phi(e,x) \end{cases} \end{cases}$$

est un homomorphisme.
***Démonstration** : On montre que cette expression correspond à celle donnée dans la définition 2.8. La*

démonstration est immédiate pour 0, 1 *et* $\top$. *On montre qu'il en est de même pour un couple* (e,α) *:*

$$\begin{aligned}\sum_{i=1}^{n}\Phi(e,a_i)(d_i\chi_i(x)) &= \sum_{i=1}^{n}(\sum_{x\in a_i}\phi(e,x))(d_i\chi_i(x))\\ &= \sum_{i=1}^{n}(\sum_{x\in a_i}\phi(e,x)d_i\chi_i(x))\\ &= \sum_{i=1}^{n}(\sum_{x\in a_i}\phi(e,x)\alpha(x))\\ &= \sum_{x\in Dom(e)}\phi(e,x)\alpha(x)\end{aligned}$$

On retrouve bien la formule de la définition 2.8. □

Proposition 6.6 (Concaténation avec étiquetage symbolique) *Soient deux DDD* d_1 *et* d_2*, le DDD* $d_1 \odot d_2$ *défini comme suit :*

$$d_1 \odot d_2 = \begin{cases} 0 & \text{si } d_1 = 0 \vee d_2 \equiv 0\\ d_2 & \text{si } d_1 = 1\\ \top & \text{si } d_1 = \top \wedge d_2 \not\equiv 0\\ (e, \sum_{i=1}^{n}(d_{1i}\odot d_2)\chi_{1i}(x)) & \text{si } d_1 = (e,\alpha)\\ & \alpha(x) = \sum_{i=0}^{n}(d_{1i}\chi_{1i}(x))\end{cases}$$

est le résultat de la concaténation $d_1 \cdot d_2$.
Démonstration *: Elle est calquée sur celle portant sur les homomorphismes, elle est immédiate pour* $d_1 = 0$*,* 1*, ou* $\top$ *et pour* $d_1 = (e,\alpha)$ *:*

$$\begin{aligned}(e,\sum_{i=1}^{n}(d_{1i}\odot d_2)\chi_{1i}(x)) &= \sum_{i=1}^{n}(e,(d_{1i}\odot d_2)\sum_{x\in a_{1i}}\chi_{1i}(x))\\ &= \sum_{i=1}^{n}\sum_{x\in a_{1i}}(e,(d_{1i}\odot d_2)\chi_{1i}(x))\\ &= \sum_{i=1}^{n}\sum_{x\in a_{1i}}(e,(\alpha(x)\odot d_2))\\ &= \sum_{x\in Dom(e)}(e,(\alpha(x)\odot d_2))\end{aligned}$$

On retrouve bien la définition 2.5. □

B.4 Chapitre 7

Proposition 7.3 (Composition de $\langle\ \mathsf{flat}\ ()\ \rangle$ et $\langle\ \mathsf{rebuild}\ ()\ \rangle$) *Soit un DDD* $d = (e,\alpha)$*, alors*

$$\langle\ \mathsf{rebuild}\ (e,1)\ \rangle \circ \langle\ \mathsf{flat}\ ()\ \rangle(e,\alpha) = (e,\alpha)$$

Démonstration *: On note d'abord que cette propriété n'est définie que sur les nœuds internes du DDD car on applique ce procédé uniquement dans le but d'abolir un niveau de hiérarchie pour une variable donnée. On développe l'équation :*

$$
\begin{aligned}
\langle \text{ rebuild } (e,1) \rangle \circ \langle \text{ flat } () \rangle(e,\alpha) &= \langle \text{ rebuild } (e,1) \rangle \circ \langle \text{ flat } () \rangle \sum_{i=1}^{n} e \xrightarrow{a_i} d_i \\
&= \langle \text{ rebuild } (e,1) \rangle \left(\sum_{i=1}^{n} \langle \text{ flat } () \rangle e \xrightarrow{a_i} d_i \right) \\
&= \langle \text{ rebuild } (e,1) \rangle \left(\sum_{i=1}^{n} a_i \cdot EOB \xrightarrow{\sharp} d_i \right) \\
&= \sum_{i=1}^{n} \langle \text{ rebuild } (e,1) \rangle (a_i \cdot EOB \xrightarrow{\sharp} d_i) \\
&= \sum_{i=1}^{n} \langle \text{ rebuild } (e,1) \rangle (\sum_{x \in a_i} \bullet_{j=1}^{m}(x_{ij}) \cdot EOB \xrightarrow{\sharp} d_i) \\
&= \sum_{i=1}^{n} \sum_{x_i \in a_i} \left(\langle \text{ rebuild } (e,1) \rangle (\bullet_{j=1}^{m}(x_{ij}) \cdot EOB \xrightarrow{\sharp} d_i) \right) \\
&= \sum_{i=1}^{n} \sum_{x_i \in a_i} \left(\langle \text{ rebuild } (e,x_{i1}) \rangle (\bullet_{j=2}^{m}(x_{ij}) \cdot EOB \xrightarrow{\sharp} d_i) \right) \\
&= \sum_{i=1}^{n} \sum_{x_i \in a_i} \left(\langle \text{ rebuild } (e,x_{i1} \cdot x_{i2}) \rangle (\bullet_{j=3}^{m}(x_{ij}) \cdot EOB \xrightarrow{\sharp} d_i) \right) \\
&\cdots \\
&= \sum_{i=1}^{n} \sum_{x_i \in a_i} \left(\langle \text{ rebuild } (e,\bullet_{j=1}^{m} x_{ij} \cdot 1) \rangle (EOB \xrightarrow{\sharp} d_i) \right) \\
&= \sum_{i=1}^{n} \sum_{x_i \in a_i} \left(e \xrightarrow{\bullet_{j=1}^{m} x_{ij} \cdot 1} d_i) \right) \\
&= \sum_{i=1}^{n} e \xrightarrow{a_i} d_i \\
&= (e,\alpha)
\end{aligned}
$$

Le point consistant à noter a_i sous la forme $a_i = \sum_{x \in a_i} \bullet_{j=1}^{m}(x_{ij})$ vient de la définition 4.2 et est calqué sur la démonstration de la propriété 4.11. □

Proposition 7.8 () *Soit v une valeur, on a $\langle \text{ set\_project } (\{v\}) \rangle = \langle \text{ project } (v) \rangle$ et*

$$\sum_{x_i \in e_i} \langle \text{ project } (x_i) \rangle = \langle \text{ set\_project } e_i \rangle$$

Démonstration *: La première propriété est immédiate si l'on reprend la décomposition $\alpha(x) =$*

$\sum_{i=1}^{n} \alpha_i \chi_i(x)$ *de la proposition 6.3.*

$$\langle\ \mathsf{set\_project}\ (\{v\})\ \rangle(e,\alpha) \;=\; \sum_{i=1}^{n} \begin{cases} \alpha_i & \text{si } e_i \cap s \neq \emptyset \\ 0 & \text{sinon} \end{cases}$$

soit e_v l'unique ensemble contenant v parmi les couples (e_i, α_i) qui décomposent α, on a :

$$\begin{aligned} \langle\ \mathsf{set\_project}\ (\{v\})\ \rangle(e,\alpha) &= \alpha_v = \alpha(v) \\ &= \langle\ \mathsf{project}\ (v)\ \rangle(e,\alpha) \end{aligned}$$

La seconde propriété se démontre par développement :

$$\begin{aligned} \sum_{x_i \in e_i} \langle\ \mathsf{project}\ (x_i)\ \rangle(e,\alpha) &= \sum_{x \in Dom(e)} \sum_{x_i \in e_i} \begin{cases} \langle\ \mathsf{id}\ \rangle & \text{si } x = x_i \\ 0 & \text{sinon} \end{cases} \\ &= \sum_{x_i \in e_i} \sum_{x \in Dom(e)} \begin{cases} \langle\ \mathsf{id}\ \rangle & \text{si } x = x_i \\ 0 & \text{sinon} \end{cases} \\ &= \sum_{x_i \in e_i} \sum_{x \in Dom(e_i)} \begin{cases} \alpha_i & \text{si } x = x_i \\ 0 & \text{sinon} \end{cases} \\ &= \sum_{x_i \in e_i} \begin{cases} \alpha_i & \text{si } e_i \cap x_i \neq \emptyset \\ 0 & \text{sinon} \end{cases} \\ &= \langle\ \mathsf{set\_project}\ e_i\ \rangle(e,\alpha) \end{aligned}$$

ce qu'il fallait démontrer. □

Annexe C

Promela Symbolique

Dans cette annexe, nous montrons comment simuler les délimiteurs de blocs de Promela pour réaliser un model-checker CTL symbolique en n'utilisant que les homomorphismes élémentaires énumérés au chapitre 5.

Sommaire

C.1 Introduction

Le but de cette section est de montrer comment simuler les délimiteurs de blocs en ne s'appuyant que sur les instructions élémentaires qu'ils contiennent. On suppose que les graphes de contrôles de chaque processus sont extraits, ce qui permet de traduire toute séquence ou structure de contrôle sous la forme d'un ensemble d'instructions concurrentes. L'emploi des séquences atomiques (`atomic`) et déterministes (`d_step`) impose néanmoins de particulariser les opérateurs pre et $post$ de chaque instruction comme le montre l'exemple suivant :

Exemple C.1 (Séquence atomique) *Considérons la séquence atomique de la partie gauche de la figure C.1 (les commentaires du code indiquent l'évolution du compteur ordinal du processus). La partie droite de la figure donne les états générés par l'application de l'opérateur post à partir de deux états distincts. Dans les deux cas, la valeur initiale de* `a` *est quelconque, mais suivant la valeur de* `b` *le nombre d'états conservés le long de la séquence sera différent :*

- *Dans la partie supérieure,* `b` *est initialement nul. La garde* `b==0 ;` *est franchie et seul l'état final est conservé. Les états intermédiaires (en pointillés) sont écartés.*
- *Dans la partie inférieure l'atomicité est perdue au niveau de la garde qui n'est pas satisfaite. Il faudra qu'un* autre *processus positionne la valeur de* `b` *à 0 pour que l'atomicité soit reprise. L'instruction qui suit la garde étant déterministe, il y aura un état intermédiaire qui sera écarté.*

Il ressort de cet exemple que l'obtention des successeurs immédiats d'un ensemble d'états suivant une séquence atomique doit tenir compte des éléments suivants :

```
atomic{
   a=1;     /* 0->1 */
   b==0;    /* 1->2 */
   a=2;     /* 2->3 */
}
```

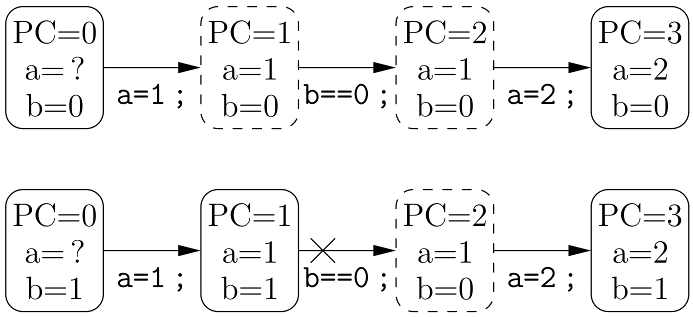


FIG. C.1 Etats générés par l'opérateur $post$ pour une séquence atomique

– les états terminaux pour lesquels l'atomicité n'a jamais été rompue.
– les états pour lesquels l'atomicité a été recouvrée.
– les états pour lesquels l'atomicité a été rompue.

Pour déterminer ces derniers, on utilisera les préconditions des instructions de la séquence pour l'application de l'opérateur $post$, on note cet opérateur $select$. Comme une séquence atomique ou déterministe peut contenir d'autres délimiteurs de blocs, chaque instruction devra aussi être pourvue d'opérateurs pre et $post$ spécifiques aux exécutions atomiques ou déterministes. On note ces opérateurs $atomic\_post$, $atomic\_pre$, $dstep\_post$ et $dstep\_pre$. Ces derniers coïncident avec les opérateurs $post$ et pre traditionnels pour les instructions élémentaires On note l'ensemble des opérateurs utiles $\mathcal{OP} = \left\{ \begin{array}{ccc} select & post & pre \\ & atomic\_post & atomic\_pre \\ & dstep\_post & dstep\_pre \end{array} \right\}$. Dans cette section, nous décrivons leurs comportements pour les délimiteurs de blocs.

C.2 Branchements et boucles

Les boucles et branchement multivoies ne font que « propager » les opérateurs de $\mathcal{OP}$ sur les différentes options qui sont considérées comme autant d'instructions concurrentes dont le comportement (atomique ou non) est le même que celui du délimiteur de bloc. La figure C.2 donne (à droite) les graphes de contrôle associés aux codes Promela (à gauche) d'un branchement et d'une boucle. Les commentaires du code indiquent les évolution des compteurs ordinaux des processus.

Ainsi pour un ensemble donné d'états S, le calcul de $\Delta(S)$ est obtenu en calculant l'union $\bigcup_i \Delta_i(S)$ des états obtenu en applicant l'opérateur (noté Δ_i) de chaque option.

Algorithme C.1 : Génération de $\Delta(S), \Delta \in \mathcal{OP}$ pour un branchement ou une boucle

Data : Un ensemble de n options
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

$ret \leftarrow \emptyset$
for $i = 1$ **to** n **do**
  $ret \leftarrow ret \cup \Delta_i(S)$
return *ret*

```
if
:: i1;      /* 0->1 */
   i2;      /* 1->2 */
   i3;      /* 2->4 */
:: i4;      /* 0->3 */
   i5;      /* 3->4 */
fi;
i6          /* 4->5 */
```

```
do
:: i1;      /* 0->1 */
   i2;      /* 1->2 */
   i3;      /* 2->0 */
:: i4;      /* 0->3 */
   i5;      /* 3->4 */
   break;   /* ?->? */
od;
i6          /* 4->5 */
```

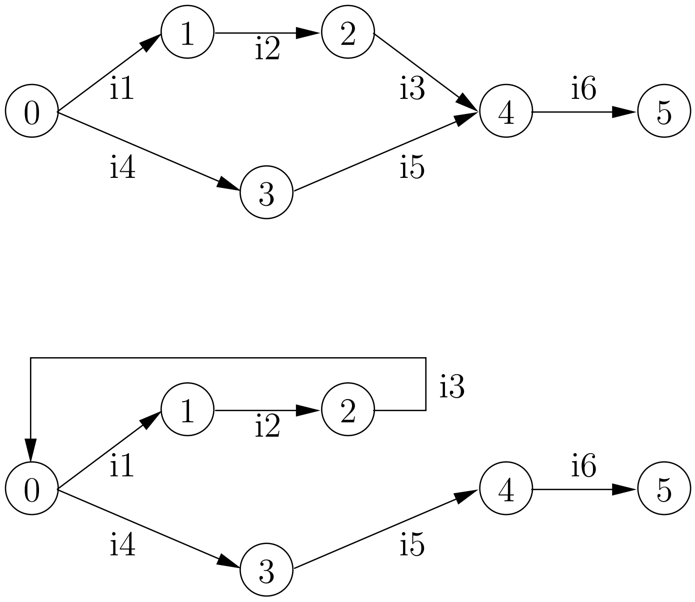


FIG. C.2 Branchements et Boucles : Graphes de contrôle

Il reste à déterminer l'évaluation des opérateurs $\Delta \in \mathcal{OP}$ sur les séquences.

C.3 Opérateurs $\Delta \in \mathcal{OP}$ pour les séquences d'instructions

Pour une séquence d'instructions quelconque, la détermination de l'opérateur $\Delta \in \{post, pre, select\}$ est semblable à celle de l'algorithme C.1 :

Algorithme C.2 : Génération $\Delta(S), \Delta \in \{post, pre, select\}$ pour une séquence

Data : Un ensemble de n instructions
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

1 $ret \leftarrow \emptyset$
2 **for** $i = 1$ **to** n **do**
3 $\quad ret \leftarrow ret \cup \Delta_i(S)$
4 **return** *ret*

La similitude avec l'algorithme C.1 est due au fait que les instructions d'une séquence sont, au même titre que les options d'une boucle ou d'un branchement, assimilables à des instructions concurrentes. Néanmoins, cet algorithme ne s'applique que pour les opérateurs $\Delta \in \{post, pre, select\}$. En effet, si la séquence apparait au sein d'une séquence déterministe (`d_step`) ou atomique (`atomic`) certains états intermédiaires devront être écartés alors que l'union de la ligne 3 de l'algorithme C.3 les conserve tous.

Pour une séquence sous le signe déterministe, on suppose que l'analyse syntaxique n'a pas détecté d'instruction indéterministe. Il n'y a pas de cumul, seuls les états finaux sont transmis en retour de fonction :

Algorithme C.3 : Génération de $dstep\_post(S)$ pour une séquence

Data : Une suite de n instructions *successives*
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

$ret \leftarrow S$
for $i = 1$ **to** n **do**
  $ret \leftarrow dstep\_post_i(ret)$
return *ret*

Dans cet algorithme, les états intermédiaires sont calculés pour atteindre la fin de la séquence mais ne sont pas conservés. Un procédé similaire est appliqué pour l'opérateur $dstep\_pre$:

Algorithme C.4 : Génération de $dstep\_pre(S)$ pour une séquence

Data : Une suite de n instructions *successives*
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

$ret \leftarrow S$
for $i = n$ **down to** 1 **do**
  $ret \leftarrow dstep\_pre_i(ret)$
return *ret*

De même que pour l'opérateur $dstep\_post$, les états intermédiaires sont calculés pour atteindre les états du début de la séquence. Néanmoins nous avons vu que, dans le cadre de la vérification utilisant les DDD, l'opérateur pre associé aux instructions élémentaires se fonde sur l'ensemble des états accessibles $Reached$. Cet ensemble devra donc être ponctuellement surapproximé pour recouvrer les états intermédiaires volontairement écartés par l'algorithme C.3.

Les séquences atomiques sont plus complexes à définir du fait de la possibilité de perdre, puis de reprendre l'atomicité. On se munit de deux ensembles :

- L'ensemble al des états pour lesquels l'atomicité est perdue : on les conserve pour une application ultérieure de l'opérateur $post$.
- L'ensemble ar des états pour lesquels l'atomicité est reprise : On les réincorpore au processus.

Les tests statuant si l'atomicité doit être reprise ou perdue n'ont pas lieu d'être à la première instruction. Ils sont effectués grâce à l'opérateur $select$.

Algorithme C.5 : Génération de $atomic\_post(S)$ pour une séquence

Data : Une suite de n instructions *successives*
Data : Un ensemble d'états où l'atomicité est perdue al
Data : Un ensemble d'états où l'atomicité est reprise ar
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

$ret \leftarrow S$
$al \leftarrow \emptyset$
$ar \leftarrow \emptyset$
$ret \leftarrow ret \cup atomic\_post_i(ret \cup ar)$
for $i = 2$ **to** n **do**
    $al \leftarrow al \cup (ret \backslash select_i(ret))$
    $ar \leftarrow select_i(S)$
    $ret \leftarrow ret \cup atomic\_post_i(ret \cup ar)$
return $ret \cup al$

Pour l'opération réciproque, on se munit des mêmes états intermédiaires que pour le calcul de l'opérateur pre sur une séquence déterministe. Pour chaque état intermédiaire, il faut statuer s'il est consécutif à une rupture d'atomicité ou non. Pour celà, on s'appuie sur l'ensemble des états accessibles $Reached$, si un état intermédiaire en fait partie, alors il correspond au premier état d'une séquence atomique *reprise*, et donc qu'il appartient à l'ensemble des états prédécesseurs *immédiats* de S.

Algorithme C.6 : Génération $atomic\_pre(S)$ pour une séquence

Data : Une suite de n instructions *successives*
Data : Un ensemble d'états où l'atomicité est perdue al
Input : Un ensemble d'états accessibles $Reached$
Input : Un ensemble d'états S
Output : Un ensemble d'états résultat ret

1 $ret \leftarrow \emptyset$
2 $al \leftarrow \emptyset$
3 **for** $i = n$ **down to** 1 **do**
4     $ret \leftarrow ret \cup atomic\_pre_i(ret \cup S)$
5     $al \leftarrow al \cup (ret \cap Reached)$
6     $ret \leftarrow ret \backslash al$
7 **return** $ret \cup al$

Dans la ligne 4 on calcule les prédécesseurs suivant le même principe que pour une séquence déterministe. Cependant, on ajoute à la séquence de calcul les états de l'ensemble S car il peut contenir les états *finaux* de séquences atomiques interrompues. Parmi les états intermédiaires calculés, certains font partie des états accessibles. Dans ce cas, ils correspondent aux états *initiaux* d'un séquence et correspondent à une reprise d'atomicité. Ils doivent donc être conservés (ligne 5) et retirés du calcul séquentiel (ligne 6) car ce sont des prédécesseurs *immédiats* de S.

Annexe D

Programmes Promela utilisés

Dans cette annexe, nous donnons quelques exemples de programmes qui ont servi aux comparaisons des performances aux chapitres 5 et 7. La plupart ont été générés à l'aide de programmes C.

Sommaire

D.1 Systèmes en Anneau

Ces systèmes comportent tous n processus communicant au travers de n canaux ainsi qu'un processus d'initialisation (init) à deux états. Chaque processus $p_i, i \in \mathbb{Z}/n\mathbb{Z}$ accède uniquement aux canaux i et $i+1$. L'écriture de tels processus en Promela est paramétrable suivant les valeurs de i et n. Un dîner de deux philosophes est encodé de la façon suivante :

```
/* Generated File of 2 philosophers */

chan f0=[1]of {int};
chan f1=[1]of {int};

active[1] proctype phil0() {
    int tmp;
    none :
    if
    ::f0?tmp;goto lft;
    ::f1?tmp;goto rgt;
    fi;
    lft:
    if
    ::f1?tmp; goto both;
```

```
    fi;
    rgt:
    if
    ::f0?tmp; goto both;
    fi;
    both:
    atomic {
        f1!tmp;f0!tmp;goto none;
    }
}

active[1] proctype phil1() {
    int tmp;
    none :
    if
    ::f1?tmp;goto lft;
    ::f0?tmp;goto rgt;
    fi;
    lft:
    if
    ::f0?tmp; goto both;
    fi;
    rgt:
    if
    ::f1?tmp; goto both;
    fi;
    both:
    atomic {
        f0!tmp;f1!tmp;goto none;
    }
}

init {
    atomic{
    f0!0;
    f1!0;
    }
}
```

La variable tampon `tmp` instanciée par chaque processus peut être déclarée comme globale pour réduire le nombre de nœuds des DDD « à plat ».

Le système de l'élection sur un anneau utilise une topologie similaire. Les processus ont un comportement plus complexe et les jetons (de valeurs différentes) sont répartis aléatoirement : ce qui conduit à $n!$ configurations *initiales* possibles. Le code ci-dessous correspond au deuxième processus d'un système à trois électeurs. L'étiquette n est déterminée par l'incrémentation concurrente de tmp, elle sera comparée à la valeur du jeton t lu dans le canal entrant.

```
/* Generated File of a 3 nodes ring*/

int tmp;
chan f0=[1]of {int};
chan f1=[1]of {int};
chan f2=[1]of {int};

...

active[1] proctype n1() {
    int n;
    int t;
    d_step {
    n=tmp;
    tmp++;
    }
    begin :
    if
    ::f2!n -> goto check;
    ::f1?[_] -> goto check;
    fi;
    check :
    f1?t;
    if
    ::atomic{(t==n); t=0; goto leader;}
    ::atomic{(t<n); t=0; f2!n; goto check;}
    ::atomic{(t<n); t=0; goto check;}
    ::atomic{(t>n); f2!t; t=0; goto loose;}
    fi;
    loose :
    do
    ::atomic{f1?t;atomic{f2!t;t=0;}}
    od;
    leader :
    tmp=4;
}

...
```

Le protocol Sliding Window est directement inspiré de [Hol97b], il lui a été ajouté la possibilité d'instancier n processus en anneau :

```
chan q0 = [6] of {byte};
chan q1 = [6] of {byte};
```

```
active[1] proctype p0()
{
    byte NextFrame;
    byte AckExp;
    byte FrameExp;
    byte r;
    byte s;
    byte nbuf;
    byte i;
      do
      ::
         ((nbuf<3));
         nbuf = (nbuf+1);
         atomic {
         q1!NextFrame;
         q1!((FrameExp+3)%(3+1));
         }
         NextFrame = ((NextFrame+1)%(3+1));

      ::
         atomic {
         q0?r;
         q0?s;
         }
         if
         ::
            ((r==FrameExp));
            /*printf('MSC: accept 0\n',r);*/
            FrameExp = ((FrameExp+1)%(3+1));

         ::
            else;

         fi;
         do
         ::
            (((((AckExp<=s)&&(s<NextFrame))
                ||((AckExp<=s)&&(NextFrame<AckExp)))
                ||((s<NextFrame)&&(NextFrame<AckExp))));
            nbuf = (nbuf-1);
            AckExp = ((AckExp+1)%(3+1));

         ::
```

```
                else;
                break;

            od;

        ::
            timeout;
            NextFrame = AckExp;
            /*printf('MSC: timeout\n');*/
            i = 1;
            do
            ::
                ((i<=nbuf));
                atomic {
                q1!NextFrame;
                q1!((FrameExp+3)%(3+1));
                }
                NextFrame = ((NextFrame+1)%(3+1));
                i = (i+1);

            ::
                else;
                break;

            od;

        od;
}
```

D.2 Bakery et Peterson

Ces systèmes se distinguent par des expressions de complexité croissante. Le code ci-dessous donne le deuxième processus (sur un ensemble de trois) pour l'algorithme du boulanger :

```
bool choosing[3];
int number[3];
int max;
...
active[1] proctype p1() {
int j;
    choosing[1]=true;
    atomic{max=max+1;number[1]=max;}
    choosing[1]=false;
    do
```

```
    ::(j<1) -> (!choosing[1]);
        ((number[j]!=0)
        || ((number[j]<number[1])
        ||((number[j]==number[1])&&((j <1)))));
        j++;
    ::(j==1)-> break;
     od;
    number[1]=0;
}
...
```

De même pour l'algorithme de Peterson :

```
int Turn[3];

int q0;
int q1;
int q2;
...
active[1] proctype p1() {
int j;
    do
    ::(1) -> {j=1;
      do
      ::(j==3)->break;
      ::else ->{
        q1=j;
        Turn[j]=1;((Turn[j]==1)||((q0>=j)&&(q2>=j)));
        j++;
      }
    od;
    }
    /*Critical section*/
    q1=0;
    od;
}
...
```

D.3 Réseau SPIN

Le code Promela des routeurs SPIN a été fourni par Hervé Charlery qui a procédé à des validations sur des modèles dégradés à deux et quatre routeurs. Ce code a été intégré à un générateur de code Promela pour tester les performances de notre model-checker sur des systèmes à plus de quatre routeurs et pour exploiter la hierarchie du système par des déclarations structurées :

```
typedef MOT {
    short adsource;
    short addest;
    short tag;
};

bool active_all;
chan poss_out0_0 = [1] of {bit};
...
chan poss_out11_3 = [1] of {bit};

short sortie_up20;
...
short sortie_up311;
chan x0_d0_spin_data_in = [1] of {MOT};
...
chan x11_u1_spin_data_out = [1] of {MOT};

chan chan_sync0 = [1] of {bit};
chan chan_sync2 = [1] of {bit};
chan chan_sync4 = [1] of {bit};
chan chan_sync6 = [1] of {bit};
...

active[1] proctype proc_port_in0_1()
{
MOT mess;
  bit reserve;

  short sortie_down;
(active_all==1);
  bit jeton = 1;

  do
::
x0_d1_spin_data_in?mess;


    if
    :: (((mess.addest / 2) % 4) == 0) ->
    /* Traitement des ports DOWN. */
        sortie_down = (mess.addest / (1)) % 2;

        if
```

```
        :: (sortie_down == 0) ->
          poss_out0_0?jeton;
          do
          ::
                x0_d0_spin_data_out!mess;

            if
            :: (mess.tag == 2) -> atomic{
                poss_out0_0!jeton;
                goto SUITE_DOWN;
                }
            :: else ->
                x0_d1_spin_data_in?mess;
            fi;
           od;
        :: (sortie_down == 1) ->
          poss_out0_1?jeton;
          do
          ::
                x0_d1_spin_data_out!mess;

            if
            :: (mess.tag == 2) -> atomic{
                poss_out0_1!jeton;
                goto SUITE_DOWN;
                }
            :: else ->
x0_d1_spin_data_in?mess;
            fi;
           od;
        fi;

        SUITE_DOWN :
skip;

    :: else ->
    /* Traitement des ports UP. */
...
```

Annexe E

Perspectives

Cette annexe présente brièvement des perspectives de recherches dans la continuité des travaux présentés dans cette thèse.

Sommaire

E.1 Amélioration de la bibliothèque [DDD]

E.1.1 Cache de calcul : Saturation et Split

Le principe de saturation consiste à modifier l'ordre d'application des traitements élémentaires pour réduire le nombre de DDD intermédiaires de calcul. Son utilisation remet en cause le fonctionnement du cache de calcul destiné à accélérer les calculs en évitant toute redondance. En effet, la saturation « locale » (algorithme 7.3.1), qui s'effectue par l'application d'un *unique* homomorphisme sur le DDD principal, interdit l'utilisation du ramasse-miettes. Il en résulte une plus grande occupation de l'espace mémoire (de l'odre de 50%) mais les calculs sont effectués avec une incomparable rapidité. Ceci laisse supposer que le compromis entre l'occupation mémoire et le temps de calcul que procure le cache est désormais déséquilibré, et nous suggérons une politique de cache moins systématique, qui puisse être paramétrée, par exemple, selon l'opérateur appliqué ou le niveau de hiérarchie concerné.

L'algorithme Split induit des calculs et des nœuds dans le cache superflus. Il est tout à fait possible, en lieu et place de l'application de l'algorithme de marquer le DDD comme nécessitant une différenciation tout en conservant le résultat de celle-ci. Lors de calculs ensemblistes, la différenciation pourra se propager sur certains opérandes conduisant ainsi à une utilisation raisonnée de split. Ce principe laisse aussi entrevoir la possibilité d'une approche hiérarchique du cache de calcul.

E.1.2 Homomorphismes « Multilinéaires »

Les « homomorphismes multilinéaires » répondent au problème de la surreprésentation des DDD à la suite de mise à plat et de différenciations. Leur définition reprendra celle des homomorphismes sur les nœuds (hiérarchiques) internes (proposition 6.5) ,

$$\Phi(e,\alpha) = \sum_{i=1}^{n} \Phi(e,e_i)(\alpha_i \chi_i(x)) \text{ avec } \Phi(e,e_i) = \sum_{x \in e_i} \phi(e,x)$$

en décomposant la somme $\Phi(e,e_i) = \sum_{x\in e_i} \phi(e,x)$ sous la forme d'un ensemble de p couples (f_j, ϕ_j), ($f_j \neq \emptyset, \phi_j \neq 0$) qui vérifie :
- $\sum_1^p f_j = e_i$
- $\forall(j,k), j \neq k, (f_j \cap f_k = \emptyset) \wedge (\phi_i \neq \phi_k)$

Ce mode de décomposition est identique à celui utilisé pour la décompositon de la fonction α effectuée au chapitre 6.

L'apport des homomorphismes multilinéaires, par rapport à notre méthode d'abolition de hiérarchie par concaténation, pourrait s'avérer décisif. En effet, ils résolvent le problème de la multiplicité des représentations qui est l'unique contrepartie, par rapport à l'étiquetage explicite des arcs, relevée au cours de nos expérimentations.

E.1.3 Homomorphismes inductifs localement composables

Au chapitre 5 nous avons fait allusion à des homomorphismes réalisant la composition de plusieurs homomorphismes en un seul parcours. Ces derniers ont été définis de façon *ad hoc*, et dans le cas d'homomorphismes modifiant des variables différentes (par exemple une variable affectée, et un compteur ordinal), la création d'une composition revient généralement à multiplier par deux le nombre d'homomorphismes à définir.

Une manière plus élégante de générer des compositions d'homomorphismes serait de définir la fonction $\phi(e,x)$ non plus sous la forme d'un unique DDD résultant (ou un homomorphisme propagé), mais sous la forme d'une composition locale de trois traitements élémentaires :

$$\phi(e,x) = \phi_{comp}(e,x) \circ (\phi_{loc}(e,x)) \cdot \phi_{prop}(e,x)$$

Où :
- ϕ_{loc} est une fonction indiquant le couple (e', x') résultant de l'application de l'homomorphisme (e,x). Par exemple pour l'homomorphisme $\langle$ setCst (var, cst) $\rangle$,

$$\phi_{loc}(e,x) = \begin{cases} e \xrightarrow{x} 1 & \text{si } e \neq var \\ e \xrightarrow{cst} & \text{sinon} \end{cases}$$

– ϕ_{prop} est une fonction indiquant l'homomorphisme à propager vers les strates inférieures. Dans le cas de l'homomorphisme $\langle$ setCst (var, cst) $\rangle$ nous aurons :

$$\phi_{prop}(e,x) = \begin{cases} e\langle \textsf{ this } \rangle & \text{si } e \neq var \\ e\langle \textsf{ id } \rangle & \text{sinon} \end{cases}$$

– ϕ_{comp} est un homomorphisme qui se calcule après l'évaluation de ϕ_{loc} et ϕ_{prop}. Son utilité intervient principalement dans le cas de l'homomorphisme $\langle$ downExpr $(s, expr)$ $\rangle$:

$$\begin{aligned} & \langle \textsf{ downExpr } (s, expr \neq cst) \rangle(e,x) \\ = & \begin{cases} \langle \textsf{ up } (e,x) \rangle \circ \langle \textsf{ downExpr } (s, expr_{e:=x}) \rangle & \text{si } e \in expr \\ \langle \textsf{ up } (e,x) \rangle \circ \langle \textsf{ downExpr } (s, expr) \rangle & \text{sinon} \end{cases} \end{aligned}$$

Dans un tel cas, on note :

$$\begin{aligned} \phi_{loc}(e,x) &= 1 \\ \phi_{prop}(e,x) &= \begin{cases} \langle \textsf{ downExpr } (s, expr_{e:=x}) \rangle & \text{si } e \in expr \\ \langle \textsf{ downExpr } (s, expr) \rangle = \langle \textsf{ this } \rangle & \text{sinon} \end{cases} \\ \phi_{comp}(e,x) &= \langle \textsf{ up } (e,x) \rangle \end{aligned}$$

Naturellement, ce cadre contient le précédent (il suffit de remplacer ϕ par ϕ_{prop}), mais dans certains cas, la composition $\phi \circ \phi'$ pourra s'exprimer sous la forme suivante :

$$\phi(e,x) \circ \phi'(e,x) = (\phi \circ \phi)'_{comp}(e,x) \circ ((\phi_\circ \phi)'_{loc}(e,x)) \cdot (\phi_{prop} \circ \phi'_{prop})(e,x)$$

La condition d'une telle transformation porte sur les variables modifiées, et un cadre théorique très strict (surtout si $\phi_{comp} \neq \langle \textsf{ id } \rangle$) doit être élaboré pour assurer sa correction.

L'avantage de ce changement de définition est la création d'homomorphismes contenant *plusieurs* ensembles de traitements élémentaires, locaux ou à propager, réalisables en un seul parcours.

Bien entendu ce cadre, s'il est adopté, devrait être compatible avec celui des homomorphismes multilinéaires.

E.2 Perpective pour la Modélisation

Dans cette section nous présentons une perspective de recherche à moyen terme pour la réalisation de systèmes décrits suivant une sémantique voisine de Promela mais tirant pleinement parti de l'approche hiérarchique.

E.2.1 L'exemple des philosophes

Le cas du dîner des philosophes est un problème purement académique dont l'utilité principale en model-checking est d'observer le comportement d'un outil pour des systèmes massivement parallèles et de comparer différentes approches sur un système commun.

Il est cependant probable que ce modèle tombe en désuétude en raison des expérimentations de [TM04] qui menèrent à la représentation de l'espace des états accessibles d'un système comportant $2^{10,000}$ philosophes.

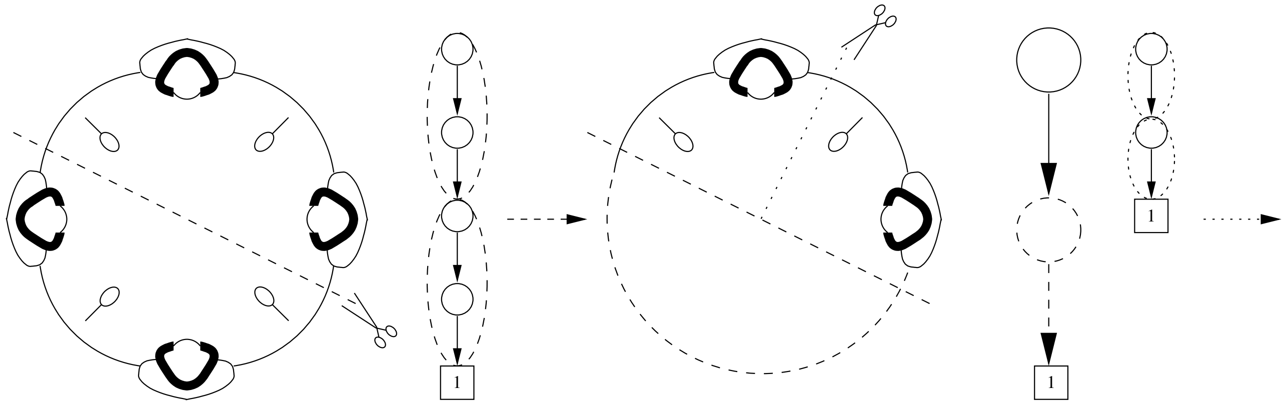


FIG. E.1 Symétries d'un système de 2^n philosophes

La figure E.1 montre comment les propriétés de symétrie du système ont été exploitées pour aboutir à un résultat aussi impressionnant. Sur la partie gauche de la figure nous avons représenté une table de quatre philosophes ainsi que le DDD les encodant, en associant un couple cuiller-philosophe par strate.

En raison de la symétrie axiale du modèle, il est autorisé de diviser ce système en deux sous-systèmes de deux philosophes (partie droite de la figure) eux mêmes divisibles en deux sous-systèmes d'un seul philosophe. La profondeur du DDD hiérarchique n'est plus alors que de deux nœuds (un par sous-système) dont les arcs sortants sont étiquetés par *un* DDD représentant un système de deux philosophes, lui même représentable par un DDD hiérarchique etc.

Ce principe dichotomique qui réduit la complexité d'un algorithme d'un ordre de grandeur n'est certes pas nouveau mais ne peut être exploité pour des systèmes Promela. En effet, la seule possibilité de définition hiérarchique des composants Promela est d'utiliser des types structurés : Les variables partagées et les processus doivent figurer sur le même niveau de hiérarchie. Dans le cas des philosophes, il serait possible de définir n structures, représentant chacune un niveau de hiérarchie, qui seraient modifiables par un unique processus régissant *l'ensemble* des philosophes :

```
typedef struct philosopher{...};

typedef struct philo_1{
philosopher phi_1;
philosopher phi_2;
}

typedef struct philo_2{
philo_1 phi_1;
philo_1 phi_2;
}
...
active[1] proctype activate_philosophers(){...}
```

Un tel mode de programmation Promela ne répond plus aux objectifs de simplicité de modélisation que nous nous sommes fixés. En revanche s'il était possible de structurer les processus de la sorte, il nous serait possible d'obtenir des performances comparables à celles de [TM04] tout en conservant un langage de modélisation accessible.

Dans la section suivante nous montrons comment nous pourrions profiter de ce découpage pour la vérification effective du réseau SPIN.

E.2.2 Application au réseau SPIN

Le réseau SPIN jouit de propriétés de symétries comparables à celles du dîner des philosophes.

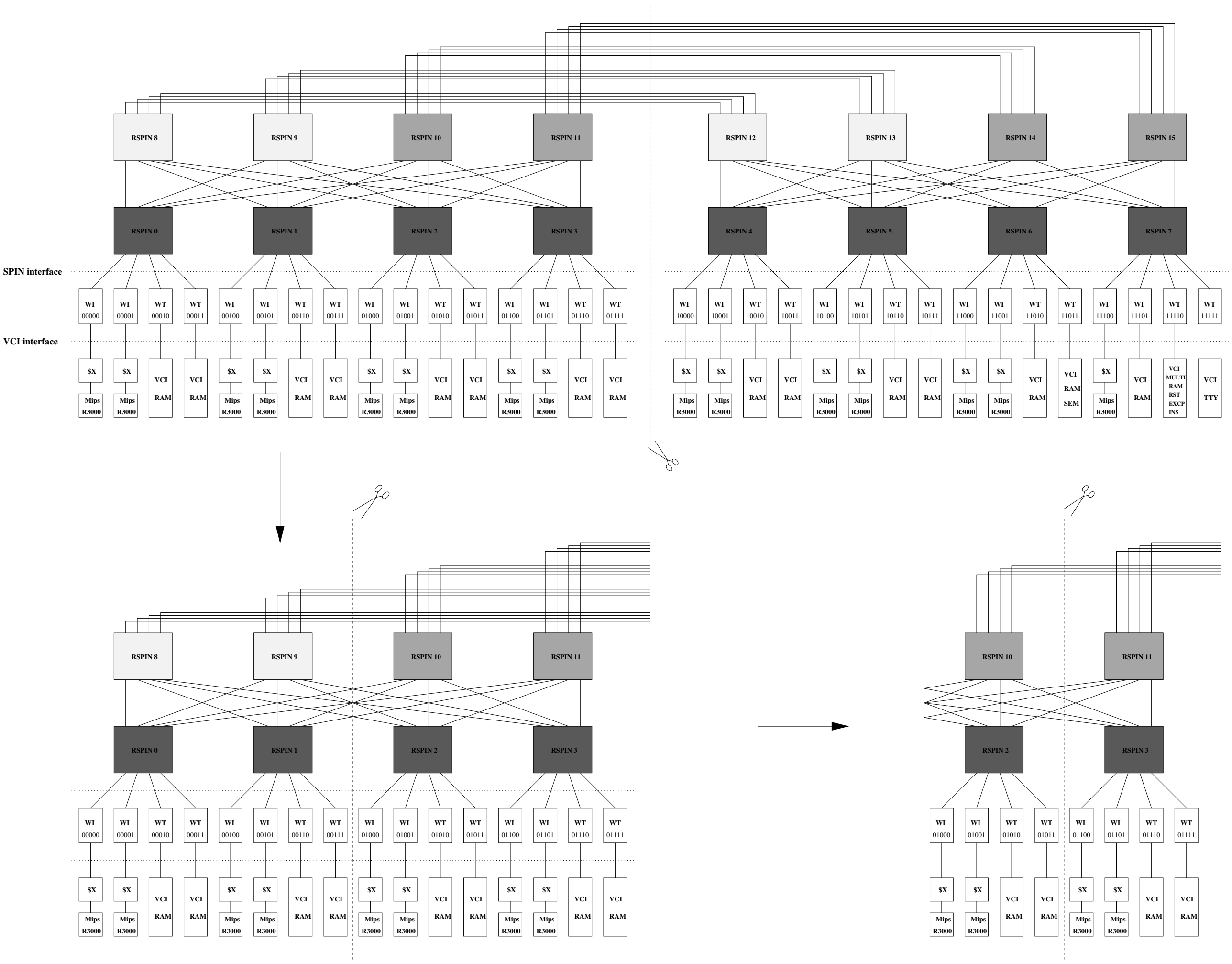


FIG. E.2 Exploitation des symétries dans le réseau SPIN

Si l'on reprend le modèle réellement implanté, on observe un premier axe vertical de symétrie entre deux sous-réseau comme le montre la partie supérieure de la figure E.2. En reprenant le principe de regroupement hiérarchique décrit avec le système des philosophes, on passe d'une représentation d'un réseau de $2 \times N$ processus à une représentation de deux sous-réseaux (gauche et droite) de N processus encodant des motifs identiques.

En réappliquant ce procédé (partie inférieure de la figure E.2) sur les sous-réseaux, la représentation comporte cette fois des sous-systèmes à quatre processus, que notre outil peut appréhender (voir section 8.2.4). De plus, un autre axe vertical de symétrie peut être exploité et si l'on sépare les abonnés du réseau proprement dit, on dispose d'un axe horizontal de symétrie.

Mieux, nous avons vu dans la section 8.2.2 que chaque routeur était en fait modélisé par un ensemble de ports pour pouvoir acheminer des messages d'origines différentes. Dans le cas du réseau SPIN original ce nombre de ports est de huit : Quatre ports ("up") pour acheminer les paquets issus des strates supérieures et quatre autres ("down") pour les paquets issus des strates inférieures.

En poussant à l'extrême le principe de regroupement par symétrie, on pourra modéliser un routeur à l'aide d'un processus encodant deux sous-ensembles ("up" et "down") de quatre ports eux-mêmes décomposables en ensembles de deux fois deux ports.

E.2.3 Définition de modèles hiérarchiques

Comme nous l'avons vu dans la section dans la section E.2.1 le langage de modélisation hiérarchique devra pouvoir définir des processus hiérarchiques ou des structures comportant des processus. Ceci suppose de pouvoir étendre la portée de certaines variables à un ou plusieurs niveaux supérieurs de hiérarchie.

D'autre part, tous les systèmes dont la taille était paramétrable (tous hormis le protocole ZCSP et la plate-forme ANI) ont été implémentés en Promela à l'aide de générateurs de fichiers programmés en C. Cette méthode présente l'avantage de pouvoir simuler l'instanciation paramétrée de processus (voir section 3.4) mais constitue aussi une rupture avec notre objectif de simplicité de modélisation. Aussi, il serait souhaitable de définir des macros remplissant cette fonction ou d'inclure l'instanciation multiple (pas nécessairement dynamique) et paramétrable de processus dans la sémantique du nouveau langage.

Enfin, nous avons choisi Promela car sa sémantique contient celles de certaines chaînes de conception de haut niveau comme Disydent. A partir du moment où l'on souhaite changer en partie la sémantique du langage de description des modèles, l'élaboration d'un langage commun pour l'implémentation réelle et la vérification (quitte à prévoir plusieurs niveaux d'abstraction) serait grandement souhaitable.

Bibliographie

[AC88] André Arnold and Paul Crubille. A linear algorithm to solve fixed-point equations on transition systems. *Inf. Process. Lett.*, 29(2) :57–66, 1988.

[ACG+03] Adrijean Adriahantenaina, Herve Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino. *SPIN : a Scalable, Packet Switched, On-chip Micro-network*. Proceedings of the Design Automation and Test in Europe Designers' Forum 2003 (DATE 2003), Munich, March 2003.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. Technical report, Carnegie Mellon, 1995.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BED03] Vincent Beaudenon, Emmanuelle Encrenaz, and Jean-Lou Desbarbieux. Design validation of zcsp with spin. In *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, pages 102–110. IEEE Computer Society, 2003.

[BHSV+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS : a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC '90 : Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45, New York, NY, USA, 1990. ACM Press.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8) :677–691, 1986.

[Bry95] Randal E. Bryant. Binary decision diagrams and beyond : Enabling technologies for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243. IEEE CS Press, Los Alamitos, 1995.

[Bud] `http://sourceforge.net/projects/buddy`.

[Cad] Cadence smv :
`http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.

[CBM90a] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Formal boolean manipulations for the verification of sequential machines. In *EURO-DAC '90 : Proceedings of the conference on European design automation*, pages 57–61, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[CBM90b] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, London, UK, 1990. Springer-Verlag.

[CC04] Ming-Ying Chung and Gianfranco Ciardo. Saturation now. In *QEST '04 : Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 272–281, Washington, DC, USA, 2004. IEEE Computer Society.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2 : An opensource tool for symbolic model checking, 2002.

[CEPA+02] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data Decision Diagrams for Petri Net Analysis. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 101–120. Springer-Verlag, 2002.

[CES86] 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.

[CFZ95] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *International Conference on Computer Aided Design (ICCAD '95)*, pages 159–163, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[CGP99] Edmund M. Clarke, Orna Grumberg, and A. Peled. *Model Checking*. MIT Press, 1999. CLA e 99 :1.

[CLS99] Gianfranco Ciardo, Gerald Luettgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. Technical report, 1999.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation : An efficient iteration strategy for symbolic state-space generation. *Lecture Notes in Computer Science*, 2031 :328– ? ?, 2001.

[CMZ+93] E. M. Clarke, K. L. McMillan, X. Znao, Fujiia Fujiia, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In ACM-SIGDA ; IEEE, editor, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 54–60, Dallas, TX, June 1993. ACM Press.

[CS02] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths, 2002.

[CTM05] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, 2005.

[CY05] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In Dominique Borrione and Wolfgang Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2005.

[DDD] Bibliothèque ddd :
`http://ddd.lip6.fr`.

[Dis] DIgital SYstem Design ENvironmenT, an open framework for the co-design of embedded systems :
`http://www-asim.lip6.fr/recherche/disydent/`.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, Becker Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In Michael Lorenzetti, editor, *Proceedings of the 31st Conference on Design Automation*, pages 415–419, New York, NY, USA, June 1994. ACM Press.

[dSv+00] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi : application modeling for signal processing systems. In *Proceedings of the 37th conference on Design automation*, pages 402–405. ACM Press, 2000.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181. Springer-Verlag, 1980.

[GDDW02] A. Greiner, E. Dreyfus, J.L. Desbarbieux, and F. Wajsburt. A Secured protocol for high speed interconnexion networks providing a remote DMA communication primitive. *Internal report of LIP6*, 2002.

[GH93] Patrice Godefroid and Gerard J. Holzmann. On the verification of temporal properties. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *PSTV*, volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, 1993.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Hol97a] G.J. Holzmann. State Compression in Spin. Twente University, The Netherlands, April 1997.

[Hol97b] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5) :279–295, May 1997. Special issue on Formal Methods in Software Practice.

[Hol98] Gerard J. Holzmann. An analysis of bitstate hashing. *Form. Methods Syst. Des.*, 13(3) :289–307, 1998.

[HP94] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[HPA01] Denis Hommais, Frédéric Pétrot, and Ivan Augé. A practical tool box for system level communication synthesis. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 48–53. ACM Press, 2001.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[KSR92] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In IEEE, editor, *Proceedings of the European Conference on Design Automation*, pages 43–47, Washington, March 16–19 1992. IEEE Computer Society Press.

[LNA99] Jørn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In *CAV'99 : Computer Aided Verification* . Lecture Notes in Computer Science, 1999.

[LS92] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 608–613. IEEE Computer Society Press, 1992.

[MC99] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In Donatelli, Susanna and Kleijn, Jetty, editors, *Lecture Notes in Computer Science : Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA*, volume 1630, pages 6–25. Springer-Verlag, june 1999.

[MCC$^+$90] J. C. Madre, O. Coudert, M. Currat, A. Debreil, and C. Berthet. The formal verification chain at BULL. In *EuroASIC*. Paris, 1990.

[Min93] S.i. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In ACM-SIGDA ; IEEE, editor, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, Dallas, TX, June 1993. ACM Press.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. Sixth Int Conf. Computer Aided Verification (CAV94)*, pages 377–390, 1994.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[PTK03] Hong Peng, Sofiène Tahar, and Ferhat Khendek. Comparison of SPIN and VIS for protocol verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2) :234–245, February 2003.

[R. 93] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE /ACM International Conference on CAD*, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3) :733–749, 1985.

[SMA] Stochastic model checking analyzer for reliability and timing (SMART) : `http://www.cs.ucr.edu/~ciardo/smart/`.

[SMV] Symbolic model verifier (SMV) : `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[Tak03] Sami Taktak. Modélisation d'un bus en anneau pour analyser les interblocages. Rapport de TER, Université Pierre et Marie Curie (Paris VI), 2003.

[Tak04] Sami Taktak. Construction d'un vérificateur de modèle symbolique pour des systèmes décrits en promela. Rapport de Stage de DEA ASIM, Université Pierre et Marie Curie (Paris VI), 2004.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, June 1972.

[TM04] Yann Thierry-Mieg. *Techniques pour le model-checking de spécification de Haut Niveau*. PhD thesis, Université Pierre et Marie Curie (UPMC, Paris 6) - LIP6, 2004.

[TP97] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design : An International Journal*, 10(2/3) :243–270, April 1997.

[VW86] M.Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2) :183–221, April 1986.

Résumé : Avec la complexité croissante des systèmes informatiques se pose la question de la mise en œuvre de méthodes automatiques pour leur vérification formelle. Parmi ces méthodes, le *model-checking* se fonde sur l'exploration exhaustive du comportement d'un système. Plus celui-ci sera complexe, plus cette exploration se traduira par une explosion combinatoire de l'espace des états du système. Diverses approches ont été proposées pour résoudre ce problème, notamment les *méthodes symboliques* qui sont basées sur une représentation compacte d'ensembles d'états.

Depuis les travaux de R.E. Bryant et la définition des Diagrammes de Décision Binaires (BDD, [Bry86]), de nombreuses représentations en DAG d'espaces d'états ont vu le jour, parmi celles-ci, on trouve les Diagrammes de Décision de Données (DDD, [CEPA+02]), qui procurent une représentation compacte d'ensemble d'états et sont pourvus de mécanismes de parcours définis *localement* pour la réalisation des modifications sur ces états et des opérations ensemblistes. Parallèlement, les travaux de G.J. Holzmann [Hol97b] ont abouti à la création de l'outil de vérification SPIN, basé sur des méthodes énumératives explicites, pour des systèmes décrits en langage Promela. Ces systèmes sont proches de ceux qui sont utilisés pour la synthèse de haut niveau [HPA01].

Nous proposons une approche de vérification symbolique de systèmes matériels décrits dans un sous-ensemble du langage Promela. La représentation symbolique d'ensembles d'états est basée sur les Diagrammes de Décision de Données qui évitent de décrire le système au niveau booléen. Nous présentons d'abord la sémantique du langage Promela ainsi que les DDD puis les mécanismes mis en œuvre pour la vérification de propriétés de logique CTL. Les conclusions tirées de cette première étape conduisent à proposer l'utilisation des Diagrammes de Décision d'Ensembles (SDD, [TM04, CTM05]) pour améliorer les performances de la vérification automatique. Nous montrons que, bien que pourvus d'une implémentation et d'une terminologie différentes, *il se prévalent du même formalisme que les DDD* tout en procurant un étiquetage *symbolique* des arcs de la structure. Nous expérimentons ensuite cette approche sur des systèmes académiques et sur des systèmes issus d'applications industrielles. Ces expérimentations corroborent nos premiers résultats : les SDD couplés aux méthodes de saturations constituent une alternative sérieuse pour la vérification de systèmes à fort degré de concurrence. Nous proposons des perspectives de recherche pour améliorer encore la verification de tels systèmes mais également pour introduire le concept de hiérarchie *dans la description du système*.
Mots clefs : Model-checking Symbolique, Diagrammes de Décision de Données, Systèmes Matériels, Promela

Abstract : With the increasing complexity of computing systems comes the problem of automatic methods for their formal verification. Among these methods, *model-checking* works on an exhaustive exploration of the system's behaviour which presents a combinatorial explosion as the system's complexity grows. Various approaches where proposed to solve this problem, like *symbolic methods* that handles compact state-space representation.

Since R.E. Bryant studies and Binary Decision Diagram's definition (BDDs, [Bry86]), many tree-based state-space representation were proposed. Among them, Data Decision Diagrams (DDDs, [CEPA+02]), which procures a compact state-space representations and uses locally-defined traversal mechanisms to express states alteration and set operators. In an other context, G.J. Holzmann studies [Hol97b] led to the model-checker SPIN, based on explicit enumerative methods for systems written in Promela. These systems are close from the ones that are used for high-level synthesis [HPA01].

We propose an approach of symbolic model-checking of Promela systems, based on Data Decision Diagrams. First, we present Promela semantics and DDDs formalism, and describes mechanisms used for CTL properties' verification. The conclusion of this first approach led us to a twin one, based on a hierarchical DDDs, called Set Decision Diagrams (SDDs, [TM04]). We show that despite differents terminology and implementation (arcs are labelled with set of states), SDDs are formally nothing more than DDDs. We experiments this approach on classical and industrial systems and suggest new research outlook to prevent the lacks we find out during our studies.
Keywords : Symbolic Model-Checking, Data Decision Diagrams, Hardware Systems, Promela