THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PIERRE & MARIE CURIE

DISCIPLINE
SCIENCES DE L'INGÉNIEUR

ÉCOLE DOCTORALE
EDITE : INFORMATIQUE, TÉLÉCOMMUNICATION ET ÉLECTRONIQUE DE PARIS (ED 130)

PRÉSENTÉE PAR
ALEXANDRE HAMEZ

POUR OBTENIR LE GRADE DE
DOCTEUR DE L'UNIVERSITÉ PIERRE & MARIE CURIE

SUJET DE LA THÈSE
GÉNÉRATION EFFICACE DE GRANDS ESPACES D'ÉTATS

SOUTENUE LE 08 DÉCEMBRE 2009 DEVANT LE JURY COMPOSÉ DE

FABRICE KORDON	DIRECTEUR	PROFESSEUR À L'UNIVERSITÉ PIERRE & MARIE CURIE (PARIS 6)
YANN THIERRY-MIEG	ENCADRANT	MAÎTRE DE CONFÉRENCES À L'UNIVERSITÉ PIERRE & MARIE CURIE (PARIS 6)
DIDIER BUCHS	RAPPORTEUR	PROFESSEUR À L'UNIVERSITÉ DE GENÈVE
PATRICE MOREAUX	RAPPORTEUR	PROFESSEUR À L'UNIVERSITÉ DE SAVOIE
BÉATRICE BÉRARD	EXAMINATEUR	PROFESSEUR À L'UNIVERSITÉ PIERRE & MARIE CURIE (PARIS 6)
ALEXANDRE DURET-LUTZ	EXAMINATEUR	MAÎTRE DE CONFÉRENCES À L'ÉCOLE POUR L'INFORMATIQUE ET LES TECHNIQUES AVANCÉES (EPITA)
FRANÇOIS VERNADAT	EXAMINATEUR	PROFESSEUR À L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE TOULOUSE

Table des matières

Table des matières **2**

1 Introduction générale **5**
1.1 Model checking . . . 7
1.2 Positionnement . . . 8
1.3 Plan . . . 10

I *Model checking* parallèle et réparti **11**

2 Problématiques et état de l'art **13**
2.1 Problématique générale . . . 13
2.2 État de l'art . . . 17
2.3 Performances dans la littérature . . . 33
2.4 Synthèse et positionnement . . . 37

3 Répartition et parallélisation du calcul d'un espace d'états **41**
3.1 Choix de conception . . . 41
3.2 Gestion de la répartition . . . 43
3.3 Vérification de propriétés de sûreté . . . 44
3.4 Architecture . . . 45
3.5 La bibliothèque LIBDMC . . . 48
3.6 Intégration avec GreatSPN . . . 49
3.7 Expérimentations . . . 50
3.8 Conclusion . . . 63

II Saturation automatique **67**

4 État de l'art et problématiques **69**
4.1 Diagrammes de décision . . . 69
4.2 Hierarchical Set Decision Diagrams : SDD . . . 71
4.3 Génération d'un espace d'état . . . 77
4.4 Saturation . . . 82

4.5 Problématiques 88

5 Saturation automatique **91**
5.1 Propagation et domaines d'application des homomorphismes . . . 92
5.2 Homomorphismes remarquables 99
5.3 Réécriture des homomorphismes 103
5.4 Expérimentations 109
5.5 Systèmes de transition instantiables 111
5.6 Conclusion 112

6 Conclusion générale **115**
6.1 Contributions 115
6.2 Perspectives 116

Bibliographie **119**

A Interfaces de la bibliothèque LIBDMC **131**

CHAPITRE 1

Introduction générale

La vérification de systèmes informatiques est une nécessité de par leur ubiquité. Leur démocratisation dans tous les domaines, surtout les plus critiques (avionique, transports, médical, etc.) impose que l'on puisse leur accorder la plus grande confiance. L'impact d'un système se révélant défectueux peut se mesurer en vies humaines ou en pertes financières colossales. Il s'agit donc d'un véritable enjeu auquel il faut répondre par des techniques de vérification performantes : les utiliser au plus tôt dans la conception d'un système permet d'éviter les coûts qu'impliquerait la détection d'une erreur grave de ce système pendant la phase d'exploitation.

Dès lors, il faut fournir à leurs concepteurs des moyens de vérification et de validation efficaces. Ces moyens existent, mais il est impératif qu'ils puissent prendre en charge des systèmes de très grande taille. En effet, les grands projets informatiques se mesurent aujourd'hui en millions de lignes de code et font souvent interagir des centaines d'intervenants. De ce fait, il ne suffit pas seulement de fournir de nouvelles techniques de vérification, mais il faut être en mesure de les rendre utilisables sur des systèmes réels.

Nous nous focalisons sur les systèmes répartis qui possèdent plusieurs composantes interagissant et s'exécutant en parallèle. Il s'agit d'une architecture de plus en plus répandue, mais leur indéterminisme intrinsèque en fait des systèmes complexes à vérifier. Cette vérification peut prendre plusieurs formes :

Tests. C'est la forme de vérification la plus simple et la plus répandue. Il s'agit d'écrire des jeux de tests qui ont pour but de vérifier qu'un système respecte un ensemble de propriétés en l'exécutant dans un environnement de tests. Ils décrivent les comportements attendus des composantes du système testé.

Ils sont généralement appliqués à deux niveaux : aux composants élémentaires du système et à l'assemblage de ces composants. L'inconvénient majeur de cette technique est sa non-exhaustivité : les tests couvrent rarement l'ensemble

du système. Cela signifie que même si les tests se déroulent correctement, il est possible qu'un comportement du système non couvert par les tests ne respecte pas les propriétés recherchées.

Simulation. Cette technique utilise la spécification du système pour en explorer différents chemins d'exécution. À chaque pas de l'exécution, on vérifie que le système respecte les propriétés respectées. Au même titre que les tests, cette technique présente l'inconvénient de ne pas être exhaustive. Ses principaux avantages sont d'être performante et d'être applicable très tôt dans le cycle de vie d'un système.

Vérification par preuve. Elle consiste à décrire un système informatique sous la forme d'axiomes. Les propriétés sont exprimées à l'aide de théorèmes qu'il faut prouver afin de valider l'architecture du système. On s'aide pour cela d'assistants de preuves tels que Coq [Coq] ou PVS [ORS92]. Cette méthode permet de vérifier de manière paramétrique un système : une propriété vérifiée reste vraie tant que les contraintes entre paramètres sont vérifiées. De plus, elle permet la vérification de systèmes infinis. Toutefois, il s'agit d'une technique difficile à mettre en œuvre et qui demande de solides compétences, et de ce fait reste difficile à diffuser.

Model checking. Au contraire de la vérification par preuve, le *model checking*[1] n'est pas une technique de vérification paramétrique. En revanche, c'est une technique plus simple à mettre en œuvre. Le principe est de générer l'ensemble des états accessibles par un système, son **espace d'états**, à partir de sa modélisation. Cette dernière en spécifie le comportement au moyen de formalismes tels que les *UML State Machines* [HN96] ou encore les réseaux de Petri [GV03]. Il est ensuite possible de déterminer si des états rencontrés ne vérifient pas la propriété recherchée à l'aide de techniques basées sur la logique temporelle. De nombreux outils existent déjà tel que SPIN [Hol97], NuSMV [CCGR99], VIS [BHSV+96], ou encore GreatSPN [CFGR95] pour ne citer que les plus connus.

La facilité d'utilisation du *model checking* provient du fait qu'il s'agit d'un processus automatisable dès lors que l'on possède un modèle formel et des propriétés à vérifier. L'autre avantage est qu'il fournit automatiquement un diagnostic, sous forme de contre-exemple, à l'utilisateur en cas de non-respect d'une propriété.

L'attribution du prix ACM Turing 2007 à Clarke, Emerson et Sifakis pour leurs travaux sur le *model checking* montre que cette technique a atteint une maturité propre à la rendre utilisable dans un contexte industriel. Par exemple, l'outil SCADE [Est] est très répandu dans le domaine de l'avionique.

1. Bien qu'étant des termes anglais, *model checking* et *model checker* sont si courants dans le domaine de la vérification que nous conserverons ces dénominations.

1.1 Model checking

Si le *model checking* présente de grandes facilités d'utilisation pour les ingénieurs, il n'en reste pas moins qu'il fait face à un problème majeur : l'**explosion combinatoire** du nombre d'états calculés [Val96] : même un « petit » système pourra produire un espace d'états qu'il serait impossible de stocker dans la mémoire d'un ordinateur.

Une grande part des travaux s'effectuant autour de cette technique consiste donc à chercher à repousser au plus loin les limites que pose cette explosion combinatoire. Les travaux décrits ici vont dans ce sens.

De nombreuses méthodes ont été mises au point pour faire face à l'explosion combinatoire. Citons entre autres :

Réduction d'ordre partiel. [Val90, God95, VAM96, Poi96]

Le principe général est d'éviter de stocker des états que l'on considérera inutiles sous certaines conditions. Par exemple, un système peut passer par une succession d'états pour invariablement aboutir à un même état. Si les états intermédiaires ne présentent pas d'intérêt par rapport à la vérification des propriétés recherchées, alors il ne sont pas conservés et ainsi la place qu'ils auraient pris en mémoire est économisée.

Techniques à base d'abstractions. [CGL94, CGJ+00]

Lorsque un modèle à analyser est trop gros, des abstractions sont faites dans l'optique d'utiliser un certain type d'analyse. Elles ont pour but de conserver des propriétés intéressantes tout en rendant possible l'analyse, puisque le modèle est réduit. Les vérifications se font ensuite sur ce modèle abstrait.

L'approche *CEGAR* (*Counter-Example Guided Abstraction Refinement*) permet de partir d'une abstraction grossière qui sera raffinée si les contre-exemples générés ne peuvent êtres réalisés sur le système concret. Comme l'abstraction est un sur-ensemble du comportement, si on ne peut plus produire de contre-exemple, alors le système concret est correct, par rapport à la propriété recherchée. Depuis son introduction, cette approche a eu beaucoup de succès.

Techniques à base de graphe quotient. [HJJJ85, CDFH91, ID93, CEJS98b]

Les états d'un système réparti présentent souvent de fortes similarités entre eux, appelées symétries. Cela signifie qu'il est possible de les stocker plus intelligemment en ayant une représentation ensembliste des états qui représenteront chacun une grande quantité d'états réels. Si bien souvent cette technique est économe en mémoire, elle est en revanche gourmande en ressources de calculs. On peut voir cette technique comme une « compression » d'espaces d'états.

Diagrammes de décision. [Bry86, BCM+90, PRCB94]

Les diagrammes de décision sont une famille de structures de données qui ont l'avantage de pouvoir stocker de manière unique les parties communes des données encodées. Or les éléments d'un espace d'états présentent très souvent de fortes similarités. De ce fait, l'usage des diagrammes de décision permet d'obtenir un gain considérable en espace mémoire. Toutefois, ils s'avèrent difficiles à manipuler et peuvent devenir inefficaces dans le cas d'utilisations impropres.

Tout comme la technique précédente, les diagrammes de décision peuvent être vus comme une manière de compresser un espace d'états. On parle alors de model checking symbolique (nous conserverons cette terminologie dans la suite de ce mémoire).

Calcul parallèle et réparti. [SD97, LS99, GMS01]

Les ordinateurs munis de plusieurs processeurs comme les *clusters*[2] sont de plus en plus répandus. Ils permettent d'obtenir une grande puissance de calcul et une grande quantité de mémoire à un prix maintenant raisonnable. Le *model checking* étant avide de ces ressources, il est naturel de chercher à marier ces deux domaines. Cette union est difficile : certains algorithmes de *model checking* ne sont pas nécessairement transposables au monde du calcul réparti.

Les travaux présentés dans ce manuscrit traitent de la génération d'espaces d'états. Ils exploitent ces deux dernières méthodes : le calcul parallèle et réparti, ainsi que les diagrammes de décision.

1.2 Positionnement

Les travaux présentés dans ce manuscrit proposent deux types de solutions pour repousser au plus loin la barrière de l'explosion combinatoire. Nos contributions ont donc pour but de générer efficacement de très grands espaces, tout en proposant des abstractions aux utilisateurs de ces contributions pour qu'ils puissent en tirer profit sans difficulté.

1.2.1 Calcul parallèle et réparti

Dans un premier temps, nous utilisons les ressources des *cluster* et des machines multi-processeurs pour fournir aux *model checkers* une puissance de calcul et une quantité de mémoire accrue. Deux objectifs sont visés : fournir un cadre de travail générique pour répartir les *model checkers* et exhiber les conditions nécessaires et suffisantes pour que cette répartition soit efficace.

Avec la fin sans cesse annoncée de la loi de Moore, les constructeurs parient de plus en plus sur les machines multi-processeurs. À terme, il ne faudra donc

2. À nouveau, ce terme anglais est si courant que nous le conserverons pas la suite

plus seulement compter sur l'augmentation des fréquences pour espérer voir les processus de vérification existants aller plus vite.

Il faut donc adapter nos algorithmes à ce nouveau monde. Il ne s'agit pas d'une tâche aisée : maints algorithmes n'ont pas été conçus pour le parallélisme. Ou alors, les programmes déjà écrits ne respectent pas les règles de bonne écriture pour le parallélisme.

Les *clusters* sont eux-aussi de plus en plus répandus. Une grande quantité de machines disponibles implique bien évidemment plus de puissance et de mémoire, mais il faut prendre en compte le réseau. Cela est une tâche encore plus ardue que l'adaptation aux multi-processeurs, car il faut cette fois échanger des données entre chaque machine. Les algorithmes de vérification qui nécessitent de nombreuses interactions au sein d'une grande quantité de données ne se prêtent pas bien à ce schéma qui induit de nombreuses communications entre toutes les machines.

1.2.2 Diagrammes de décision

Ensuite, nous appliquons des techniques d'accélérations aux diagrammes de décisions, en nous basant sur une technique existante. Nous la généralisons et en automatisons l'utilisation, jusque là impossible, pour faciliter l'écriture de *model checkers* efficaces.

L'article à l'origine des diagrammes de décision [Bry86] est un des plus cités dans le domaine informatique. Et pour cause : le *model checking* n'est pas la seule technique à profiter des avantages de cette structure de données. Dès lors qu'il est nécessaire de stocker de grandes quantités d'informations qui peuvent assimilables à des vecteurs, alors les diagrammes de décision ont de grandes chances d'apporter une réponse plus que satisfaisante.

Depuis l'introduction des *Binary Decision Diagrams* (BDD) dans [Bry86], de nombreuses variations ont vu le jour, la plupart du temps pour répondre à un besoin spécifique. Ces diagrammes de décision peuvent être vus selon deux axes : la structure de données en elle-même et la manipulation de cette structure.

Les *Data Decision Diagrams* [CEPA+02] (DDD) ont apporté une technique de manipulation souple et puissante en proposant les homomorphismes. Ces derniers sont des opérations clairement définies et clairement structurées. Ils introduisent une souplesse qui permet de séparer les données manipulées des opérations, ce qui n'était pas le cas avant.

Les *Hierachical Set Decision Diagrams* [CTM05] (SDD), une évolution des DDD, ont ensuite apporté un mécanisme jusque là inusité dans les diagrammes de décision : la hiérarchie. Cet ajout permet de faire « coller » au mieux les diagrammes de décision aux spécifications modernes qui proposent des modèles de composition, souvent basés sur la hiérarchie.

La combinaison de la hiérarchie et des homomorphismes a permis d'obtenir d'excellentes performances. Toutefois, dans le contexte des MDD [SHMB90] (assez semblables aux DDD par ce qu'ils peuvent représenter), est apparu un nou-

veau mécanisme d'évaluation appelé « saturation » qui offre des gains en temps d'exécution et en consommation mémoire conséquents. Ce mécanisme peut être généralisé : il repose sur le principe que tous les éléments d'un diagramme de décision ne sont pas nécessairement modifiés ensembles.

Cependant, sa mise en œuvre pour les MDD [CLS01a] est à la fois complexe et non générale, les auteurs s'étant cantonnés à la génération d'un espace d'états.

1.3 Plan

Les deux approches sont disjointes. Elle seront donc par la suite traitées en deux parties distinctes.

La première partie présente le ***model checking* parallèle et réparti**. Nous y établissons l'état de l'art (chapitre 2), puis présentons une architecture dédiée au *model checking* réparti explicite et nous présentons ses performances (chapitre 3). La conclusion de cette partie présente les enseignements tirés de cette architecture et les conditions favorables au *model checking* réparti.

La deuxième partie présente le travail effectué sur la manipulation d'espace d'états encodés symboliquement. Un état de l'art présente les SDD et la technique de saturation (chapitre 4). Ensuite nous exposons les mécanismes nécessaires à la mise en place de la **saturation automatique** (chapitre 5).

Enfin, nous synthétisons ces deux approches et présentons les pistes à suivre pour des travaux futurs (chapitre 6).

Première partie

Model checking parallèle et réparti

CHAPITRE 2

Problématiques et état de l'art

Ce chapitre présente le *model checking* parallèle et réparti explicite ainsi que les problèmes et les solutions rencontrés dans la littérature s'y rapportant.

2.1 Problématique générale

Le *model checking* peut-être ramené à la vérification d'une propriété P sur un système Φ : $\Phi \models P$. Plusieurs types de propriétés peuvent être vérifiés tels que les propriétés de sûreté ou de causalité.

Les propriétés de sûreté sont les plus simples à vérifier car elles n'utilisent pas la logique temporelle et peuvent se ramener à la construction d'un espace d'états (on parle alors « d'accessibilité »). Elles n'ont cependant pas le même pouvoir d'expression que les propriétés de causalité. Ces dernières sont vérifiées au moyen de logiques temporelles. Toutefois, vérifier des propriétés de sûreté est très souvent suffisant : on peut détecter des interblocages, vérifier des bornes ou encore qu'une section critique est bien respectée par les processus y accédant. De plus, sous certaines conditions, il est possible de ramener des problèmes de logique temporelle à de l'accessibilité en utilisant des observateurs [AS89].

Vérifier une propriété de sûreté revient à parcourir les états que peut atteindre un système, et, pour chaque état rencontré, vérifier si la propriété est respectée. Il s'agit donc dans l'idée d'un parcours de graphe (en largeur ou en profondeur), comme présenté dans l'algorithme 2.1 : on part d'un ensemble d'états initiaux S_0 que nous plaçons dans un ensemble d'états $nouveaux$ à visiter. Tant qu'il existe des états qui n'ont pas encore été calculés, on détermine leurs successeurs en utilisant les relations de transition qui décrivent le système Φ, tous les états n'ayant encore jamais été visités seront à leur tour placés dans l'ensemble $nouveaux$. La vérification de propriétés de sûreté est bien ramenée à un problème d'accessibilité : elle s'effectue au moment de la construction de l'espace d'états. Les contributions des travaux présentés ici se placent dans ce contexte d'accessibilité.

Algorithme 2.1 : Génération de l'espace d'états d'un système Φ et vérification de la propriété p

```
1  pour chaque s ∈ S_0 faire
2      nouveaux ← nouveaux ⋃ s
3      S ← S ⋃ s
4  tant que nouveaux ≠ ∅ faire
5      nouveaux \ {s}
6      pour chaque s' ∈ successeurs(s) faire
7          si s' ∉ S alors
8              nouveaux ← nouveaux ⋃ {s'}
9              S ← S ⋃ {s'}
10             si ¬(s' ⊨ P) alors
11                 retourner (Φ ne vérifie pas P)
12 retourner (Φ vérifie P)
```

Si vérifier des propriétés de sûreté est simple, il n'en reste pas moins qu'il faut être en mesure de générer les espaces d'états des systèmes vérifiés. Or les systèmes réels que l'on cherche à vérifier sont bien souvent trop complexes pour être calculés à l'aide d'un seul ordinateur.

Ce problème est particulièrement observable lors de la vérification de systèmes répartis asynchrones. En effet, ceux-ci sont constitués de multiples composants dont les espaces d'états sont relativement indépendants. L'espace d'états global est donc presque le produit cartésien de tous ces sous-espaces, amenant fatalement à un très grand nombre d'états qu'il serait impossible de tous stocker.

Un des buts du *model checking* est donc de faire face à ce problème. De nombreuses solutions ont été mises au point : réduction d'ordre partiel, utilisation des symétries, utilisation de *clusters* de machines et/ou de machines multiprocesseurs, etc.

L'utilisation d'un *cluster* de machines apporte deux avantages :

1. Un gain en quantité de mémoire
2. Un gain en puissance de calcul

Plus de mémoire implique de pouvoir stocker plus d'états, et peut-être ainsi de pouvoir terminer un calcul qui n'aurait pas été possible sur une seule machine.

Cependant, des techniques comme l'utilisation des symétries [CDFH91, CEJS98a] ont permis de compresser les espaces d'états à tel point qu'une seule machine peut contenir des espaces d'états potentiellement gigantesques. Cela a toutefois un coût : le gain en mémoire se fait aux prix d'un besoin en puissance de calcul lui aussi énorme.

Par exemple, la vérification de l'intergiciel PolyORB [HVP+05] n'a pu être effectuée pour un trop grand nombre de *threads* modélisés, et ce non pas pour un

manque de mémoire, mais pour un manque de temps de calcul. L'usage d'un *cluster* permettrait donc cette fois-ci de rendre possible le calcul dans un temps raisonnable.

Dans le cas où l'on cherche seulement à gagner en puissance de calcul, il n'est pas impératif de répartir la génération de l'espace d'états. En effet, seul compte les ressources de calcul disponibles, donc le nombre de processeurs. Comme aujourd'hui il est courant de trouver des machines dotées de plusieurs processeurs, il est intéressant de paralléliser les procédures de *model checking*, pour profiter de ces architectures modernes.

Remarque
Par la suite, le terme « répartir » fera toujours référence au fait d'utiliser un ensemble de machines reliées par un réseau au sein d'un *cluster* alors que « paralléliser » indiquera le fait d'utiliser l'ensemble de processeurs d'une machine partageant la même mémoire physique.

Les figures 2.1 et 2.2 illustrent ces deux derniers points. Dans le cas de la figure 2.1 représentant le nombre d'états pour le fameux modèle du dîner de philosophes en fonction du nombre de philosophes : on assiste à une croissance exponentielle du nombre d'états.

La figure 2.2 montre le temps nécessaire à la génération du graphe d'accessibilité de l'intergiciel PolyORB en fonction du nombre de *threads* modélisés : il faut très rapidement plusieurs dizaines d'heure pour en finir l'analyse. Cet exemple est d'autant plus intéressant que la modélisation de PolyORB a été faite en prenant en compte au mieux les spécificités du *model checker* visé (GreatSPN) : les symétries. La raison de ce choix était d'éviter l'explosion combinatoire en fournissant au *model checker* un model dont les caractéristiques le rendraient particulièrement efficace à analyser. Si cela s'est avéré efficace pour le nombre d'états, le temps de calcul s'en est retrouvé considérablement rallongé.

La répartition du *model checking* est orthogonale vis-à-vis des autres techniques d'optimisation de *model checking*. En effet, si aujourd'hui, par exemple, conjuguer les diagrammes de décision à l'ordre partiel semble un objectif lointain, il est bien plus envisageable de répartir de telles techniques, à moins bien sûr d'algorithmes fondamentalement séquentiels et n'autorisant aucune exécution parallèle.

Parmi toutes les solutions pour combattre l'explosion combinatoire, la répartition de la génération d'un espace d'états en utilisant un *cluster* de machines semble être la plus « technique ». En effet, répartir un code existant relève plus souvent du domaine de l'ingénierie logicielle que de la recherche.

Cependant, le *model checking* présente des particularités qui rendent la tâche délicate. En effet, la découverte d'un espace d'états n'est pas régulière : le nombre de successeurs calculés pour chaque état est différent, ce qui fait qu'il est impos-

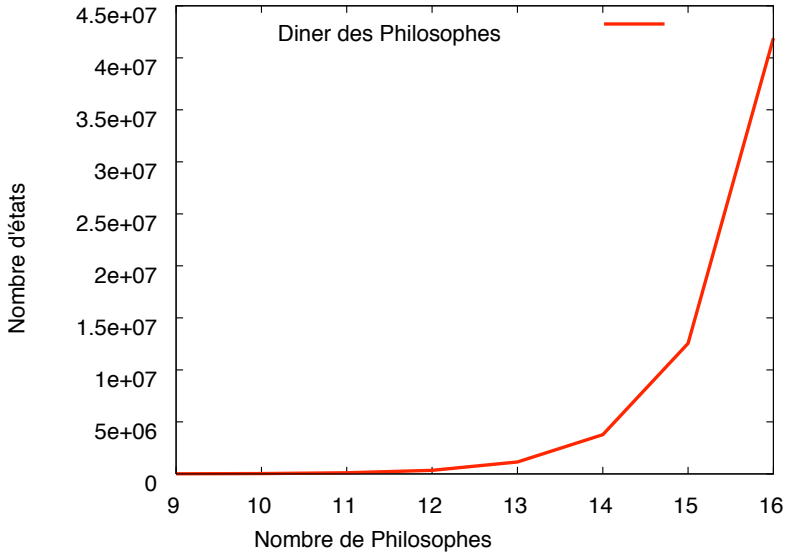


FIGURE 2.1 – Philosophes : nombre d'états en fonction du nombre de philosophes

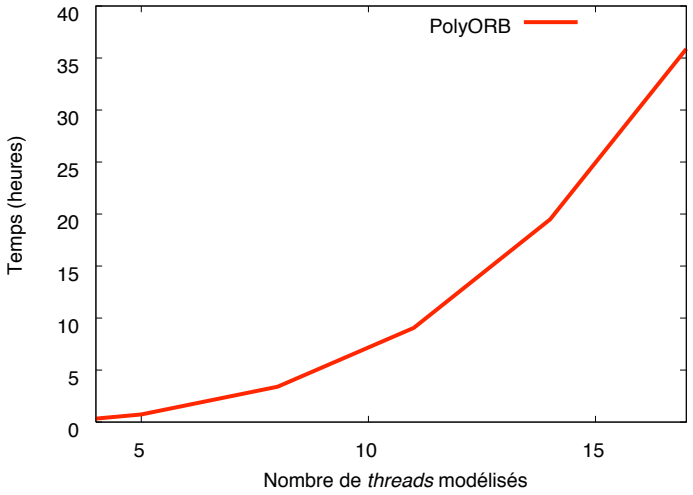


FIGURE 2.2 – Intergiciel PolyORB [HVP+05] : temps de calcul en fonction du nombre de *threads* modélisés

sible de connaître à l'avance le temps de calcul que prendra l'exploration d'un état. Cela peut poser des problèmes de famine lors d'une exécution sur un *cluster*, puisque des machines pourraient n'avoir que des états « trop faciles » à calculer. De plus, les modèles vérifiés ne se comportent pas tous de la même manière : une technique de génération pourra très bien fonctionner pour un certain type de modèle et pas pour d'autres.

L'objectif principal de la parallélisation ou de la répartition d'un algorithme est d'atteindre une **accélération linéaire**. L'accélération est la mesure qui indique le rapport entre temps de calcul séquentiel [1] et le temps de calcul réparti. Une accélération linéaire signifie que si l'on utilise n machines, alors le calcul s'effectue n fois plus vite.

Dans le cas du *model checking*, c'est un objectif difficile car le temps de calcul d'un état n'est pas constant. Il est donc difficile de s'assurer que toutes les machines sont utilisées au maximum de leurs capacités en permanence.

Un second objectif est le **passage à l'échelle**. Il s'agit d'une appréciation indiquant qu'un algorithme réparti se comporte de manière aussi satisfaisante sur 2, 10 ou 1000 machines. Pour avoir cette qualité, un algorithme réparti doit avoir plusieurs caractéristiques. Premièrement, il doit être économe en communications réseau. En effet, un nombre trop grand de messages échangés peut impliquer un écroulement du réseau ; ou encore des messages trop grands peuvent entraîner des temps de traitement trop long sur les nœuds et saturer le réseau. Ensuite, il doit éviter de s'appuyer sur des mécanismes de diffusion (*broadcast*), car ceux-ci sont rarement efficaces sur les réseaux *Ethernet*, les plus courants. Enfin, il faut que sa complexité ne dépende pas du nombre de nœuds du *cluster*.

Le *model checking* réparti doit bien évidemment satisfaire ces objectifs, comme tout calcul réparti. Mais aussi, il doit pouvoir remplir la tâche principale d'un *model checker* qui est la vérification. Cela signifie donc qu'il faudra mettre en place des mécanismes permettant d'inspecter un espace d'états réparti sur un *cluster* de machines.

La création en 2002 du *workshop Parallel and Distributed Methods in verifiCation* [2] montre bien l'intérêt grandissant de la communauté pour le *model checking* parallèle et réparti.

2.2 État de l'art

Nous présentons dans cette section les travaux de la littérature concernant la génération parallèle et répartie d'espaces d'états. Nous positionnerons les travaux présentés dans le chapitre suivant à la fin de cet état de l'art.

Nous intéressant spécifiquement au *model checking* explicite, c'est à dire n'utilisant pas de structures de données partagées telles que les diagrammes de déci-

1. Nommé ainsi pour faire opposition à la notion de calcul parallèle
2. http://pdmc.informatik.tu-muenchen.de

sion, nous ne présenterons pas ici les *model checkers* symboliques répartis. On peut toutefois évoquer [ELS07] ou encore [GHIS05].

2.2.1 Historique : choix d'une architecture répartie et parallèle et d'un modèle de communication

Répartir et paralléliser un algorithme nécessite de choisir sur quelle type d'architecture il sera exécuté. On utilise habituellement la taxinomie de Flynn [Fly72] pour décrire les architectures parallèles et réparties, Cette taxonomie est présentée au tableau 2.1.

	Instruction unique	Instructions multiples
Donnée unique	*SISD*	*MISD*
Données multiples	*SIMD*	*MIMD*

TABLE 2.1 – Taxonomie de Flynn

Le *SISD* désigne les architectures séquentielles, encore très courantes aujourd'hui où une seule instruction est appliquée à une seule donnée à la fois, ce qui correspond aux machines mono-processeur. Le *MISD* est quant à lui très peu usité. Actuellement, les architectures *SIMD* et *MIMD* sont les plus courantes pour le parallélisme. La première correspond à ce qu'on connait généralement sous le nom de calcul vectoriel : une même instruction est appliquée en parallèle sur plusieurs données différentes. La deuxième correspond au fait que l'on applique en parallèle des calculs différents sur des données différentes, ce qui peut correspondre aussi bien aux machines multi-processeurs qu'aux *clusters* de machines.

Les premiers travaux concernant la répartition d'un *model checker* se sont effectués avec des machines dédiées. Dans [CCBF94], GreatSPN a été porté sur une CM-2 (*Connection Machine 2*), une machine de type SIMD (*Single Instruction Multiple Data*). L'avantage principal d'une telle architecture est de posséder plus de mémoire qu'une station de travail habituelle, cela permet à cette implémentation parallèle de gérer plus d'états explicites. Cependant le temps de calcul est bien plus long que sur une station de travail avec un algorithme séquentiel. Cela s'explique par le fait qu'un espace d'états est une structure très peu régulière, et donc impropre à un calcul de type SIMD. Le seul intérêt d'une telle architecture est donc de posséder plus de mémoire que les machines standards, et donc de pouvoir stocker plus d'états.

Les mêmes auteurs ont ensuite essayé de déterminer l'approche la plus adaptée [CCM95] : les données sont parallélisées ou alors la transmission de données se fait par passage de messages. La première approche correspond donc au modèle de programmation *SIMD*, tandis que la seconde correspond au modèle de programmation *MIMD*. Ils évaluent ainsi l'adéquation des ces deux modèles par rapport à la structure typiquement irrégulière et imprévisible d'un espace d'états.

Il s'avère finalement que le passage de messages est plus adapté à cette structure que la parallélisation des données, ce que les auteurs confirment à nouveau dans [CCM01]. On peut noter l'utilisation des messages actifs dans [SD97], qui sont à cheval entre le passage par messages et l'invocation de procédures distantes, puisqu'ils contiennent l'adresse de la procédure à exécuter sur la machine cible.

[AKH97] propose d'utiliser une mémoire partagée sur une seule machine multi-processeurs. L'intérêt réside davantage dans le fait de disposer de ressources de calcul supplémentaires que d'accroitre la quantité de mémoire importante. Il est ainsi possible de calculer plus vite les états, mais pas d'en stocker plus.

Cette approche n'est pas incompatible avec une architecture de type MIMD, puisqu'un nœud sur un *cluster* de machines peut très bien posséder plusieurs processeurs. Il semblerait toutefois qu'il n'y ait, à l'heure actuelle, aucun model checker alliant les architectures MIMD et SIMD. Toutefois, comme beaucoup d'implémentations utilisent MPI, il est très facile de lancer plusieurs instances sur un même nœud, simulant ainsi la conjonction de ces deux approches (mais la mémoire ne sera évidemment pas partagée).

Remarque
Dans cette partie concernant le *model checking* réparti, le terme « nœud » indiquera toujours une machine au sein d'un *cluster*.

Tous les travaux menés sur la répartition du *model checking* se sont orientés vers les communications par passage de messages sur des architectures à mémoire répartie, ce qui correspond aux architectures actuelles de *clusters* de machines peu chères. La plupart des articles cités par la suite ce situent donc généralement dans ce contexte.

2.2.1.1 De la génération séquentielle à la génération répartie et parallèle

Nous avons vu le principe de la génération séquentielle d'un espace d'états en 2.1. La génération répartie suit exactement le même principe, si ce n'est qu'elle introduit la notion de **propriétaire d'un état**. En effet, comme il s'agit cette fois de répartir les calculs sur un ensemble de machines, il faut être en mesure d'affecter une portion de l'espace d'états à chaque nœud. La figure 2.3 illustre cela : on y voit par exemple que les états 1 et 2 sont affectés au nœud C du *cluster*. Une partition de l'espace d'états est effectuée, chaque sous-ensemble étant possédé par un nœud et un seul du *cluster*.

La construction répartie d'un graphe n'est pas en soi difficile. Il faut cependant déterminer si un sommet de ce graphe a déjà été rencontré ou non, ce qui nécessite un mécanisme de stockage, généralement une table de hachage. Il faut donc que ce système de stockage soit réparti. Pour ce faire, il faut utiliser une fonc-

tion de localisation permettant de savoir où doit être stocké sur le *cluster* chaque sommet du graphe.

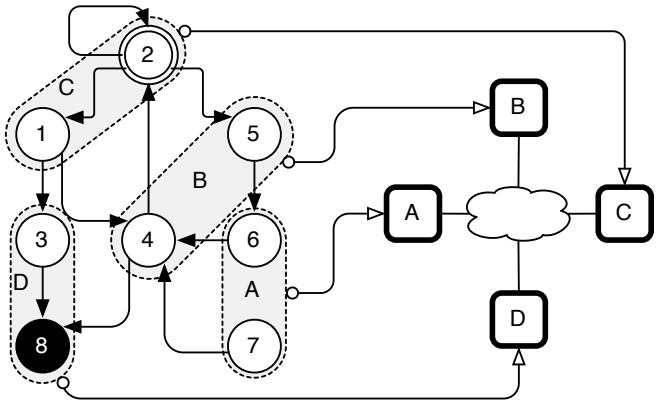


FIGURE 2.3 – Distribution des états sur un *cluster*

Ainsi lorsqu'un nœud calcule un successeur, il doit non seulement déterminer si cet état est nouveau, mais il doit aussi déterminer s'il en est le propriétaire, afin de continuer à calculer ses successeurs ou pour l'envoyer au nœud qui le possède. L'algorithme 2.2 montre cet ajout : à la ligne 7, on calcule l'identifiant du nœud possédant le successeur calculé. À la reception d'un état, il est directement placé dans l'ensemble $nouveaux$ s'il n'est pas présent dans S, autrement, il sera simplement placé dans S.

La génération répartie « casse » donc le sens du parcours. En effet, en séquentiel il peut-être en profondeur ou en largeur d'abord, alors qu'en réparti, des états peuvent arriver du réseau à tout moment pour être insérés dans la file d'attente des états à calculer.

À la vue des ces éléments, deux problèmes se posent :

- Chaque nœud est responsable d'un sous ensemble de l'espace d'états : comment s'assurer que tous les nœuds du *cluster* soient utilisés au maximum de leurs capacités en permanence ?
- Lorsqu'un nœud calcule un successeur qui ne lui appartient pas, il doit le transmettre au propriétaire : comment minimiser les communications ?

Ces deux problèmes sont sous-jacents aux diverses solutions de la littérature exposées par la suite. Par exemple, la minimisation des communications ne consiste pas seulement en une optimisation mais aussi en une contrainte forte : si la méthode de localisation du propriétaire d'un nœud générait une communication pour chaque état, alors le système matériel supportant le model checker réparti serait très vite écroulé. Ou, sans aller jusqu'à l'écroulement, un trop grand nombre de messages nuirait à la mise à l'échelle du *model checker* réparti et rendrait cette approche inutile. De la même manière, si la méthode de localisation attribuait une partie de l'espace d'états à un seul nœud de façon disproportionnée par rapport aux autres nœuds, alors sa mémoire se verrait très vite saturée,

Algorithme 2.2 : Génération répartie de l'espace d'états S d'un système Φ et vérification de la propriété p

```
1  pour chaque s ∈ S_0 faire
2      nouveaux ← nouveaux ⋃ s
3      S ← S ⋃ s
4  tant que nouveaux ≠ ∅ faire
5      nouveaux \ {s}
6      pour chaque s' ∈ successeurs(s) faire
7          id ← localisation(s')
8          si id = nœud local alors
9              si s' ∉ S alors
10                 nouveaux ← nouveaux ⋃ {s'}
11                 S ← S ⋃ {s'}
12                 si ¬(s' ⊨ P) alors
13                     retourner (Φ ne vérifie pas P)
14         sinon
15             envoyer s' au nœud id
16 retourner (Φ vérifie P)
```

empêchant ainsi à coup sûr la génération de l'espace d'états complet.

Si à première vue cet algorithme de génération semble conçu pour une répartition sur un *cluster*, il est aussi utilisable sur une machine multi-processeurs. La différence réside dans le fait que la fonction de localisation ne retourne plus un identifiant de nœud d'un *cluster*, mais celui d'un processeur ; l'envoi vers le propriétaire se réduit à une recopie dans sa file d'attente des états à explorer.

Notons que cet algorithme de génération répartie n'a aucune dépendance sur le nombre de nœuds, ce qui est une bonne caractéristique pour qu'il passe à l'échelle.

2.2.2 Répartition d'un *model checker* : problématiques

On a précédemment vu (section 2.2.1.1) le principe général de la génération d'un espace d'état de manière répartie. Cette répartition pose un ensemble de problèmes auxquels différentes solutions ont été proposées.

En premier lieu, il faut pouvoir déterminer la façon de gérer les états (leur codage, stockage et répartition) de la manière la plus efficace sans que cela n'entrave la répartition (section 2.2.2.1).

Ensuite, générant un espace d'états explicite, il faut s'assurer de son exploration exhaustive, ainsi qu'en vérifier la terminaison (section 2.2.2.2).

Quelle que soit la manière de répartir le calcul et de représenter les états, un *model checker* prend toujours un modèle de haut niveau en entrée. Il est donc na-

turel de se demander l'impact des modèles sur le comportement général (section 2.2.2.3) qu'aura le *model checker* réparti.

Enfin, l'écriture d'un *model checker* est une tâche difficile : il dépend généralement d'un formalisme et nécessite de longs travaux de recherche. En écrire un à partir de rien dans le seul but de le répartir est une perte de temps importante. Il faut donc pouvoir réutiliser l'existant (section 2.2.2.4).

2.2.2.1 Gestion des états

Pour gérer les états dans le *model checker* réparti, il faut pouvoir les coder et les stocker, comme tout *model checker* classique. On a vu qu'il faut aussi pouvoir associer à chaque état un propriétaire sur le *cluster*. Il faut donc ajouter un mécanisme de localisation des états qui renvoie l'identifiant du nœud en charge d'un état particulier.

Codage des états. Si un nœud veut pouvoir déterminer le propriétaire d'un état sans pour autant qu'il interroge chacun des autres nœuds du *cluster*, alors il doit pouvoir le faire avec les informations dont il dispose localement. Et cette information n'est rien d'autre que l'état lui-même. On voit donc l'importance que revêt la manière de décrire un état.

Il s'avère que comme les implémentations rencontrées à travers la littérature sont bien souvent des adaptations de *model checker* existant, elles ne font que réutiliser le codage disponible dans celui-ci. C'est donc un aspect assez peu traité dans la littérature. On peut toutefois noter [LV01] dans lequel ils implantent un nouveau *model checker* pour Java et dans lequel ils utilisent l'état courant des *threads* ainsi que celui des objets pour définir un état, ce qui est une extension au principe de SPIN.

Même si ce problème d'encodage est peu abordé, la distribution des états implique que leur codage soit cohérent à travers tous les nœuds du *cluster*. En d'autres termes, il faut que quel que soit un état, son codage soit indépendant de données locales à un nœud du *cluster*, auxquelles les autres nœuds ne pourraient pas accéder. Par exemple, il ne faudrait pas que l'une des composantes du codage corresponde à une entrée dans une table de hachage locale, auquel cas elle serait indisponible à l'ensemble des autres nœuds. Ou alors il faudrait mettre en place un système complexe de répartition de ces types de données afin de les rendre accessibles à tout le *cluster*.

Pour illustrer ce problème, citons la version répartie de SPIN [LS99] dont la version séquentielle utilisait directement la représentation binaire du codage des états. Bien qu'étant très efficace, ce système n'est pas directement exploitable sur un *cluster* de machines hétérogènes, aux représentations de données internes différentes. Les auteurs ont donc dû s'appuyer sur une couche *XDR*[3] afin d'encoder les états de manière universelle. Bien que n'étant pas directement lié à la manière

3. *eXternal Data Representation* : c'est un standard permettant d'encoder les données indépendamment de toute architecture matérielle

de choisir les composantes du codage des états, ce problème montre bien l'importance d'une représentation commune indépendante de toute donnée locale à un nœud. De plus, il faut que ce procédé de sérialisation soit très rapide pour ne pas nuire aux performances globales.

Stockage des états. C'est un problème qui n'est pas vraiment traité dans la littérature puisqu'il est déjà résolu dans le cadre d'un *model checker* séquentiel. Ce stockage est toujours réalisé (excepté certaines implémentations réparties, présentées par la suite, utilisant des arbres) à l'aide d'une table de hachage qui permet de rapidement déterminer si un état a déjà été généré ou non. Chaque nœud du *cluster* étant responsable de son stockage. Dans le cas d'une implémentation purement parallèle de SPIN [HB07], les processeurs partagent la même table de hachage, alors protégée par un sémaphore.

Localisation des états. Il s'agit du cœur du problème car une bonne partition de l'espace d'états doit assurer à la fois que chaque nœud possède une charge équivalente aux autres et qu'un minimum de communications auront lieu. Ainsi, si une partie voit un grand nombre de ses successeurs appartenir à une multitude d'autres parties, alors il faudra que le nœud responsable de cette partie envoie souvent des messages contenant ces successeurs aux autres nœuds.

Diverses techniques de localisation ont fait leurs apparitions dans la littérature. Ainsi sont utilisées actuellement les méthodes suivantes :

- un hachage statique ;
- un partitionnement de l'espace d'états de manière statique ou dynamique en fonction des composantes du codage d'un état ;
- des arbres équilibrés ;
- des arbres de décision.

Hachage statique Il s'agit de la méthode la plus simple : pour déterminer l'identité du nœud propriétaire d'un état, on applique une fonction de hachage sur le codage de cet état, le résultat de cette fonction correspondant à l'identité du nœud qui en est responsable. [SD97] utilise une fonction de hachage universelle. Cette classe de fonctions de hachage a pour principe de choisir aléatoirement la fonction à partir d'une classe universelle. Les auteurs montrent que, statistiquement, la distribution des états est uniforme sur l'ensemble des nœuds, assurant de ce fait une bonne répartition de la charge.

Nombreuses sont les implémentations qui utilisent cette méthode : [CGN98, MCC97, BLW01, Cia01]. Cependant, deux problèmes se présentent : une mauvaise fonction de hachage peut répartir les états de telle sorte qu'une petite partie des nœuds se retrouve en charge d'une très grande partie de l'espace d'états. Ensuite, comme le souligne [LS99], l'uniformité de la répartition n'assure pas forcément un faible nombre de communications. En effet, le hachage ne préserve pas nécessai-

rement la notion de localité, ce qui fait que, par exemple, les états de l'évolution séquentielle d'un processus modélisé se retrouvent dispersés.

Partition de l'espace d'états Cette méthode peut se faire de manière statique ou dynamique.

- *Partition statique*
  Le partitionnement est déterminé à priori et aucun changement ne peut être effectué à l'exécution. Les avantages sont la simplicité de mise en œuvre et le fait qu'aucune communication ne soit engendrée pour déterminer une politique de répartition.
  [LS99] utilise cette technique en s'appuyant sur les informations de localité que peut fournir le codage, en l'occurrence les informations relatives au statut d'un *thread* choisi. En effet, dans SPIN, l'état global du système contient une composante par processus du modèle. En partant du constat que, lors d'une interaction entre deux processus ou lors d'une action locale à un processus, peu de composantes évoluent dans le codage, une partition simple consiste à se baser sur la valeur prise par une seule de ces composantes pour déterminer le nœud responsable de l'état. Les auteurs montrent aussi qu'en utilisant une telle technique, ils réduisent le nombre de transmissions d'états entre les nœuds.
  Leurs résultats montrent clairement que le nombre de messages échangés, ainsi que le temps d'exécution, est bien plus faible dans le cas d'une fonction de partition que dans le cas d'une fonction de hachage, mais avec une répartition moins bonne. Cependant, pour atteindre cela, la fonction de partition a dû être déterminée à la main, pour chaque modèle. Les auteurs expérimentent aussi une approche intermédiaire qui consiste à appliquer une fonction de hachage sur une seule composante, amenant à une répartition des états de qualité intermédiaire.
  On retrouve un schéma de partition similaire dans [PPP07] : les auteurs utilisent le marquage des réseaux de Petri pour identifier des sous-ensembles à partir desquels déterminer la localisation des états. Tout comme [LS99], ils observent un nombre de communications réduit, mais au prix d'éventuelles mauvaises répartition de la charge de calcul. Une approche similaire est faite dans [KP04].
  La version purement parallèle de SPIN [HB07] utilise les transitions dites « irreversibles » pour créer une partition sur l'espace d'états, ces transitions étant identifiables par une analyse statique du modèle vérifié. Ce type de transition indique que ses états sources ne sont pas atteignables à partir de ses états cibles. Ainsi, cela permet d'identifier des sous-ensembles disjoints dont les calculs peuvent être réalisés indépendamment.
- *Partition dynamique*
  La partition est adaptée à l'exécution en utilisant des informations recueillies pendant le déroulement de la génération. Son principal inconvé-

nient est de générer plus de communications, mais en contre partie, la répartition est effectuée en fonction du modèle sans pour autant que l'utilisateur ait à fournir une politique de répartition. Cette méthode suppose clairement que si un ensemble d'états se trouve à un instant donné sur un nœud, il peut très bien se retrouver par la suite sur un autre nœud.
C'est l'approche utilisée dans [LV01] qui regroupe les états en classes, qui sont elles-mêmes des sous-ensembles de chaque partition. Les classes n'ont pas forcément la même taille. Ce sont ces classes qui migreront de nœud en nœud, si jamais le partionnement doit être redéfini. Cette redéfinition est typiquement nécessaire dans le cas où des nœuds viendraient à manquer de mémoire. Dans ce cas, une nouvelle fonction de partition est définie (il n'est pas précisé quels sont les critères) : les états ayant déjà été visités, mais qui appartiennent dorénavant à une autre partition sont supprimés du nœud qui en était précédemment responsable et ils sont à nouveau calculés sur le nouveau propriétaire. L'inconvénient majeur est donc que beaucoup de calculs sont susceptibles d'être refaits.
Les auteurs de [LV01] proposent deux méthodes pour créer une partition initiale du modèle : les classes sont uniformément réparties sur l'ensemble des nœuds, ou alors un nœud est initialement en charge de toutes les classes, et au fur et à mesure du calcul, ces classes seront distribuées sur le *cluster*. Au vu des résultats, la première méthode semble plus performante. Toutefois, leurs tests n'ayant été effectué que sur deux machines, la mise à l'échelle et la réelle efficacité n'est donc pas clairement établie.

Arbres équilibrés Après avoir expérimenté un hachage statique [CGN98] pour lequel il fallait définir une fonction différente pour chaque modèle, Nicol et Ciardo ont tenté de répartir l'espace d'états de manière automatique en utilisant des arbres équilibrés [NC97]. Ils justifient ce choix par deux constats : la difficulté de trouver manuellement une fonction de hachage efficace et la possibilité qu'un nœud se retrouve surchargé sans qu'il soit possible d'affecter une partie de sa charge à un autre nœud à cause de la nature statique du hachage. Ils proposent donc d'automatiser la répartition des états. Les états sont regroupés en classes qui seront les sommets de l'arbre équilibré. Chaque classe est initialement associée à un nœud du *cluster*.

Lors du calcul d'un état, son codage est comparé à la racine de l'arbre, puis, en fonction du résultat de cette comparaison, il est comparé au sous-arbre gauche ou droit, jusqu'à trouver la classe à laquelle il appartient. Il est ensuite envoyé au nœud propriétaire en fonction d'une table indiquant l'équivalence entre les nœuds et les classes. La recherche dans cet arbre suppose donc que les codages des états soient ordonnés selon un ordre total. Pour calculer un ensemble de classes initiales permettant une répartition optimale, il est effectué un premier parcours aléatoire de l'espace d'états. L'ordre total est lui aussi aléatoire : les composantes du codage sont permutées aléatoirement. Selon leurs études, cette mé-

thode préserve la localité des états.

[CM97] va plus loin que [NC97] en considérant l'espace d'états comme le produit cartésien de sous-ensembles de cet espace et en associant à chacun d'entre eux un arbre équilibré. La répartition se fait donc pour chaque sous-ensemble de l'espace d'états.

[ADK98] se base sur les réseaux de Petri, et plus spécifiquement le marquage des places : la recherche dans l'arbre équilibré se fait en utilisant un ordre total défini en fonction des marquages. Chaque nœud devient responsable d'une partie contiguë de l'ensemble des marquages potentiellement accessibles.

Arbres de décision Une autre approche utilisant des arbres est exposée dans [Sch03]. Les états sont répartis sur le *cluster* à l'aide d'un arbre de décision binaire dans lequel seuls les chemins menant à la valeur booléenne *true* sont représentés. Chaque sommet de l'arbre est coloré par un processus P_i. Un état est représenté sur une feuille et chaque processus est responsable de tous les états appartenant au sous-arbre colorié par ce processus. Les états sont donc codés à l'aide vecteurs de booléens.

L'arbre n'est vu que partiellement par chaque nœud. Un processus P_i voit donc d'un arbre la structure logique suivante :

- tous les sommets colorés par P_i ;
- tous les sommets sur les chemins qui mènent de la racine aux sommets colorés par P_i ;
- tous les successeurs des sommets colorés par P_i (si jamais ces successeurs existent).

Pour trouver à qui appartient un état, une recherche est effectuée sur l'arbre de décision. Cette recherche peut-être déclenchée par le calcul d'un état en local ou alors par la réception d'un message provenant d'un autre nœud. Cette recherche consiste à parcourir la structure logique de la racine jusqu'aux feuilles, en suivant le chemin fourni par le codage de l'état. Si ce chemin ne se termine pas sur un sommet possédé par le nœud, alors cela engendre un nouveau message vers le nœud possédant le sous-arbre. Sinon, si le chemin se termine sur une feuille locale, l'état existait déjà. Ou encore si le chemin se termine sur un sommet qui n'est pas une feuille, l'état est inséré dans la branche, ou alors le nœud peut en déléguer la propriété à un autre nœud du *cluster*.

Un intérêt de cette technique est le fait qu'un nœud peut déléguer à un autre nœud la propriété d'un état, cette délégation n'étant déclenché que lorsqu'un nœud vient à manquer de mémoire. Par contre, on peut se se retrouver dans des cas où un message est généré à chaque fois que l'on passe de sommet en sommet, pour passer la main aux responsables des sous-arbres consécutifs. De plus, le protocole présenté empêche à un nœud de refuser la propriété d'un sous-arbre, et il n'y a pas de résultats montrant la véritable efficacité de cette approche.

Les techniques basées sur une représentation des états à l'aide d'un d'arbre

présentent l'inconvénient d'un temps de recherche variable pour déterminer la localisation d'un état, lequel temps est fonction de la hauteur de ces arbres, contrairement aux fonctions de hachage ou de partition qui se font en un temps constant.

Elles présentent par contre l'avantage d'une plus grande flexibilité à l'exécution, puisqu'il est possible de définir à la volée le propriétaire d'un nouvel état et il est aussi possible d'ajouter de nouveaux nœuds dynamiquement.

En général, les auteurs des différents travaux s'accordent sur un point : la méthode de localisation doit être transparente pour l'utilisateur. Par exemple, dans le cas d'un hachage statique, l'utilisateur ne doit pas fournir cette fonction de hachage. Ainsi dans [KP04], ils ont dû adapter leur fonction de hachage en fonction du modèle analysé afin d'obtenir une répartition convenable. Ils proposent donc comme objectif de recherche d'effectuer une première passe sur le modèle afin d'en déduire une fonction de hachage adaptée. Dans leur cas, comme le formalisme utilisé est les réseaux de Petri colorés, ils proposent d'utiliser les informations fournies par les marquages des places.

[LS99] souligne aussi l'importance de ne pas laisser à l'utilisateur la charge de fournir directement la fonction permettant de localiser les états. Lorsqu'ils comparent le hachage à la partition, ils annoncent clairement que fournir une fonction de hachage est totalement empirique.

En s'appuyant sur leurs travaux de [CGN98], les auteurs de [NC97] sont les seuls à exiger de l'utilisateur la fonction de hachage. Ils supposent en effet que de dernier est le seul à avoir une connaissance approfondie de son modèle afin d'en déduire une fonction de hachage suffisamment efficace pour que chaque nœud ait une charge de travail équivalente.

2.2.2.2 Calcul réparti et parallèle des successeurs

La littérature ne s'éloigne pas de l'algorithme 2.2 : ils utilisent quasiment tous une fonction de localisation qui permet de créer une partition d'un espace d'états. Nous regardons donc ici ceux qui présentent quelque originalité par rapport à cet algorithme de génération.

Algorithme de calcul réparti. Dans [KP04], si le nœud ayant calculé un état n'en est pas responsable, alors il l'envoie à un coordinateur, qui le transmet au bon nœud. L'intérêt semble assez limité puisque il faut deux communications pour transmettre un état. Un avantage serait toutefois que chaque nœud n'a pas besoin de connaître les autres nœuds.

Lorsqu'un état est codé de telle manière qu'il devient trop coûteux de le transmettre, alors, comme le préconise [LV01], il vaut mieux envoyer le chemin qui conduit à cette état. L'inconvénient de cette méthode est bien évidemment que le propriétaire de l'état en question devra le calculer à nouveau, impliquant des pertes en ressource de calcul, puisque celui-ci a déjà été effectué. Le problème

de la taille du message se posera à nouveau si le chemin est trop grand pour être transféré.

Algorithme de calcul parallèle. [HB07] est le seul travail que nous avons trouvé dans la littérature qui s'intéresse au calcul purement parallèle d'un espace d'états. Ce travail est basé sur SPIN. L'approche est la suivante : le parcours se faisant en profondeur, dès que la profondeur dépasse un seuil, alors le parcours continue sur un autre processeur. Il n'utilise donc pas de fonction de localisation d'états.

Terminaison. Comme tout calcul, réparti ou non, il faut s'assurer de sa terminaison. Sachant que par hypothèse les systèmes vérifiés sont finis, on est assuré que le calcul s'effectuera en temps fini. Mais étant donné que les communications sont asynchrones et que chaque nœud n'a pas connaissance de l'état d'avancement de ses voisins, il n'est pas possible de détecter la terminaison sans ajout d'un mécanisme supplémentaire.

Les auteurs de [CGN98] testent deux algorithmes différents : un algorithme de Dijkstra [DFvG83] (la terminaison est détectée en utilisant des messages synchrones pour faire circuler un jeton), puis un autre de Nicol [Nic95] (à base de barrière de synchronisation), ce dernier semblant mieux passer à l'échelle. Les auteurs de [GMS01] utilisent un algorithme basé sur une topologie en anneau inspirée de [Mat87] (la terminaison est détectée en vérifiant qu'il n'y a plus de messages en transit sur le réseau, donc dans le cas du *model checking* réparti, plus d'états en circulation).

[LS99] met en place un processus en charge de détecter la terminaison. À chaque fois qu'un nœud entre en phase d'attente ou de calcul, il envoie un message au processus principal. Lorsque celui-ci détecte que tous les nœuds sont en phase d'attente, il demande une confirmation à chacun de ces nœuds.

Pour [SD97], un nœud responsable demande, après un certain temps d'inactivité, aux autres nœuds le nombre de messages qu'ils ont reçus et envoyés. Si la soustraction entre la somme des messages envoyés globalement et la somme des messages reçus globalement est nulle, et qu'il n'y a plus d'états en attente de calcul, alors le calcul est terminé. Toutefois, comme montré dans [Mat87], cet algorithme de détection est faux. Il suffit cependant de procéder à cette phase de détection une deuxième fois pour qu'il devienne correct.

2.2.2.3 Impact du modèle analysé

Il est possible de se contenter de générer l'espace d'états de manière naïve, sans se soucier de ce que le modèle peut exprimer comme propriétés. Cependant, il parait très intéressant de chercher à en extraire des informations afin d'améliorer le comportement général d'un *model checker* réparti.

Impact de la structure du modèle sur la répartition. Un formalisme servant à décrire un système réparti permet bien souvent d'exprimer des propriétés de ma-

nière explicite ou implicite. Par exemple, Promela permet de décrire explicitement les processus d'un système. Ou bien, en réseaux de Petri, une analyse de la structure permettra de fournir des invariants correspondants par exemple à des processus.

En extrapolant ces informations, on peut en déduire la proximité de certains ensemble d'états. Il est donc judicieux de profiter de cette notion de localité pour placer ces états proches sur le même nœud dans le but d'éviter d'avoir à transférer des états trop souvent.

[LV01] référence [Ler00] (malheureusement en italien) qui propose l'utilisation d'une analyse statique basée sur le graphe de contrôle de flux d'un *thread* afin de générer une fonction de partition statique appropriée au modèle.

[CCM01] utilise le marquage des places d'un réseau de Petri stochastique en les associant à des poids pour déterminer une fonction de hachage adaptée au modèle. Les places choisies le sont de trois manières différentes :

- automatiquement en se basant sur les bornes ;
- automatiquement mais en évitant que les places sélectionnées n'appartiennent au même invariant ;
- manuellement.

Il s'avère que le choix de places n'appartenant pas au même invariant ainsi que l'utilisation de nombres premiers en tant que poids est la meilleure combinaison, et assure une répartition équitable sur les nœuds, ainsi qu'un temps de calcul moindre. Il n'est cependant pas fait mention du nombre de messages transmis.

Notons qu'un des membres de l'équipe développant DiVinE [BBCŠ05], une librairie pour construire des *model checkers* répartis, a essayé de mettre en évidence des propriétés structurelles typiques d'un espace d'états [Pel04] de manière empirique en générant un grand nombre d'espaces d'états à partir de différents modèles. Bien qu'il ne s'intéresse pas à ce que ces propriétés typiques peuvent apporter à l'élaboration d'un *model checker* réparti, ces informations pourront être exploitées afin d'améliorer le calcul.

En résumé, il n'y pas eu d'études sur l'impact du modèle sur la répartition des états, mais seulement des tentatives de mise au point d'heuristiques plus ou moins sophistiquées en fonction des modèles analysés. Toutes ces heuristiques s'appuient sur la notion de localité. En effet, celle-ci permet de déterminer la proximité de deux états et utiliser cette distance pourrait s'avérer utile pour déterminer la localisation des états sur le *cluster* afin de minimiser le nombre de communications.

2.2.2.4 Indépendance de la relation de franchissement

Nous avons voulu déterminer dans quelle mesure l'architecture des *model checker* répartis tentaient de séparer la logique de répartition de celle du calcul des successeurs. Nous présentons ici les travaux qui, sous forme de bibliothèque ou simplement conceptuellement, ont cherché à séparer ces deux aspects.

Interfaces génériques. Le fait de s'abstraire du modèle de haut niveau a une incidence sur le calcul réparti. En effet, dans ce cas le *model checker* dispose de moins d'informations sur la structure du modèle à partir desquelles il pourrait faire certaines hypothèses afin d'optimiser la génération.

Dans [NC97], il est proposé l'interface suivante :

- $Initial$ retourne l'état initial du modèle ;
- $Enabled(s)$ retourne l'ensemble des événements qui peuvent être appliqués à l'état s ;
- $NewState(s,e)$ retourne l'état atteint après l'exécution de l'événement e sur l'état s ;
- $Compare(s_1,s_2)$ retourne le résultat de la comparaison entre deux états suivant un ordre total.

Le pouvoir d'expression est très faible rendant impossible à l'utilisateur d'une telle interface de spécifier des relations entre d'éventuels composants du modèle.

Citons aussi [GMS01] qui utilise OPEN/CÆSAR [Gar98], un environnement de *model checking* offrant des interfaces génériques pour divers formalismes. Toutefois, les auteurs estiment que le fait d'être indépendant du formalisme de description des modèles implique nécessairement qu'il est impossible de prendre en compte les propriétés qu'il aurait pu exprimer. Ce faisant, on perdrait des informations qui pourraient être utiles à la génération répartie afin de la rendre plus efficace.

L'indépendance vis-à-vis des modèles analysés implique de prendre en compte leurs spécificités. Ainsi, par exemple, les automates temporisés nécessitent un ordre de parcours rigoureux. Les interfaces à fournir doivent donc être en mesure de capturer le maximum d'informations des modèles de haut niveau, non pas tant pour optimiser la génération répartie, mais pour garantir la cohérence de l'espace d'états générés. L'idéal serait donc des interfaces couvrant l'ensemble des besoins, ce qui parait difficilement réalisable.

La bibliothèque DiVinE. DiVinE [BBCŠ05] se présente comme bibliothèque fournissant une plate-forme de développement pour les outils de vérification basés sur l'énumération exhaustive. Elle est programmée en C++ et utilise MPI pour l'aspect réseau. Elle est distribuée sous licence GNU GPL. Elle se veut une bibliothèque permettant aux chercheurs de concevoir, implanter et expérimenter des algorithmes pour des *model checker* répartis. Elle fournit par exemple des primitives de synchronisation sur une barrière, ou encore un mécanisme de détection de terminaison. Un outil a été réalisé avec cette librairie : il prend en entrée soit le langage DVE natif à DiVinE, soit ProMeLa [BFLW05], langage de modélisation de SPIN.

La bibliothèque Eddy. Eddy est une bibliothèque de *model checking* répartie. Dans [MPS+06], elle est présentée dans une implémentation avec Murφ. Sur chaque nœud, un *thread* est dédié au calcul de l'espace d'états, tandis qu'un autre

se charge entièrement des communications, ce qui implique donc que la génération en elle-même n'est pas parallélisée, mais seulement répartie. Elle est basée sur MPI et détecte la terminaison via un mécanisme à base de jeton.

2.2.3 Optimisations du *model checking* réparti

La génération répartie d'un espace d'états est un processus exigeant en ressources de calcul, de mémoire et de réseau. Il est donc important de chercher à optimiser ces paramètres afin d'atteindre des performances optimales.

Dans le cas des ressources de calcul, il ne s'agit pas d'économiser sur des calculs inévitables, mais plutôt de s'assurer que toutes les machines du *cluster* dédié à la vérification participent de manière active à la génération. Il est donc utile de chercher à répartir la charge de calcul de nœuds trop chargés sur d'autres oisifs (section 2.2.3.1).

Le processus de calcul de l'espace d'états génère un grand nombre de messages transitant sur le réseau, puisqu'il faut très souvent transférer un état d'un nœud l'ayant calculé à un autre qui en est le propriétaire. Pour économiser ce nombre de messages, il est possible d'utiliser un cache (section 2.2.3.2).

Toujours dans le cadre du réseau, il est tout aussi important de se poser la question de l'efficacité la technologie qui est employée pour gérer les communications (2.2.3.3).

2.2.3.1 Équilibrage de charge

L'équilibrage de charge peut s'effectuer selon deux indicateurs : la mémoire restante sur un nœud et le nombre d'éléments restants dans la file d'attente des états nouveaux dont il faut déterminer les successeurs, ce dernier indicateur correspondant plutôt à la charge processeur de la machine.

L'équilibrage de charge peut se faire à deux niveaux :

– directement au moyen de la fonction de localisation des états ;
– par l'emploi d'une couche supplémentaire de répartition de charge.

Dans l'idéal, une fonction de localisation équilibre au mieux la charge sur chaque nœud. En pratique, une telle fonction est difficile à obtenir car elle est dépendante du modèle analysé. Ainsi [CCM01] s'appuie sur le marquage des places d'un réseau de Petri, [LS99] s'appuie sur les composantes du codage des états ou encore [SD97] fait le pari qu'une fonction de hachage universelle distribuera de manière équilibrée les états. On le voit, différentes réponses ont été apportés mais ne constituent pas une réelle solution au problème. De plus, la répartition seule des états ne suffit pas, il faut aussi qu'elle assure un nombre minimum de communications.

Dans [NC97], il est proposé une solution permettant de répartir dynamiquement la charge en se basant sur des arbres équilibrés, tout comme [ADK98]. Dans celui-ci, un nœud maître surveille les autres nœuds esclaves. Si jamais un des nœuds a une utilisation mémoire supérieure à 10% de celles des autres nœuds,

alors le maître indique à chaque esclave quelle quantité de mémoire doit être transférée sur deux de ses voisins. Après ce transfert, chaque esclave indique au maître quels sont les sous-arbres de l'arbre équilibré qu'il possèdent (en réalité, ils indiquent juste les bornes inférieures et supérieures des marquages qu'ils possèdent). La répartition s'appuie donc ici directement sur la structure qui détermine à la localisation des états.

Une autre possibilité est de calculer dynamiquement une fonction de hachage en fonction du modèle. À notre connaissance, seul [KJM04] s'est attaqué au problème. La vérification est commencée sur un seul nœud et lorsque la quantité de mémoire utilisée a dépassé un certain seuil, ce nœud devient le père d'un autre nœud. La fonction de hachage est alors calculée à nouveau pour prendre en compte l'existence du nouveau nœud et tous les états sont transférés du père vers le fils. Ce mécanisme est répété jusqu'à la génération totale de l'espace d'états ou jusqu'à épuisement de la mémoire. Cependant, au dire des auteurs, la méthode ne présente quasiment que des inconvénients, et le graphe d'accessibilité n'est même pas réparti de manière uniforme.

L'adjonction d'une couche supplémentaire en charge de la répartition est proposée dans [KM05], et plus en détail dans [Kum04]. Pour ce faire, ils se basent tous deux sur [XL94, WLR93].

L'ensemble des machines est réparti selon une architecture logique en hypercube. Cette structure a pour avantage que chaque nœud possède le même nombre de liens vers les autres nœuds. Le principe est le suivant : toutes les i itérations (i étant fixé par l'utilisateur), chaque processeur échange la taille de sa file d'états en attente. Si un nœud s'aperçoit qu'il a une charge supérieure à un de ses voisins, alors il lui envoie la moitié de ses états en attente. Les résultats montrent très clairement une très bonne répartition du calcul , tous les nœuds possédant au cours du temps une file de même taille.

2.2.3.2 Utilisation d'un cache

Pour garantir le passage à l'échelle d'un *model checker* réparti il est nécessaire de minimiser le nombre de messages échangés entre les nœuds. En plus d'atteindre ce but par un choix judicieux de la fonction de localisation, l'introduction d'un cache des états envoyés permet de restreindre le nombre de communications.

Avant chaque envoi d'un état vers un autre nœud, le cache est consulté pour vérifier si cet état a déjà été envoyé. Dans le cas contraire, il est envoyé et ajouté dans le cache. [LV01] montre des gains de 10% en terme de messages envoyés, avec un léger gain en terme de vitesse. [KM05] indique un gain en de 2% à 10%. De plus, ils soulignent le fait qu'il n'est pas nécessaire d'utiliser un gros cache (de l'ordre du Ko), donc que cela n'a pas un coût prohibitif en mémoire.

[LV01] introduit aussi la notion de *children lookahead*. Cette technique consiste simplement à calculer les successeurs immédiats d'un état avant d'envoyer ce dernier vers son possesseur. Ainsi, à la réception de cet état, le nœud qui

en a la charge n'aura pas besoin de transmettre ses successeurs, puisqu'il auront déjà été calculés. Il est tout à fait possible de calculer plus d'un niveau de successeurs, au prix d'un coût supérieur en temps de calcul, pas forcément rentabilisé.

2.2.3.3 Choix des technologies réseau

Bien qu'étant un détail purement technique, le choix d'une implémentation réseau a quelques répercussions. En règle générale, MPI [GLS99] ou les sockets UNIX en TCP sont utilisés, ou encore PVM [GBD+94] de manière plus anecdotique.

Les travaux utilisant MPI ou PVM mettent en avant la simplicité et la portabilité, tandis que ceux utilisant directement les sockets UNIX évoquent un trop grand surcoût impliqué par l'utilisation de librairies de haut niveau [LS99, Jou03]. Cependant, aucune comparaison n'a été faite.

Dans [Sch03], il est fait usage du protocole UDP. Si l'on se place dans le cadre d'un réseau fiable, alors ce choix permet d'éviter les surcoûts liés au protocole TCP. Cependant, cela implique une gestion plus complexe des *buffers* et des flux de contrôle.

Cependant, tous s'accordent sur l'usage d'un *buffer* d'envoi pour agréger les états avant de les envoyer au nœud responsable [SD97, Beh02, GMS01]. Et si jamais les *buffers* ne sont pas pleins au bout d'un certain temps, ils sont tout de même envoyés afin d'éviter une situation de famine.

2.3 Performances dans la littérature

La mesure principale de performance d'un calcul réparti ou parallèle est son **accélération**, c'est à dire le rapport entre le temps nécessaire pour le calcul en réparti ou parallèle et le celui en calcul séquentiel, l'idéal étant une **accélération linéaire**.

Nous présentons donc ici les performances des outils et bibliothèques de la littérature, sous forme d'accélération quand cette information est disponible, sinon uniquement sous forme de temps.

2.3.1 Murφ [SD97]

Sur trois modèles testés, cette implémentation atteint presque une accélération linéaire (figure 2.4) : les accélérations obtenues varient entre 26.6 et 29.4, sur 32 machines. Leurs résultats confirment que l'utilisation d'un *buffer* d'états à envoyer ensemble permet de gagner significativement en temps et en nombre de messages échangés.

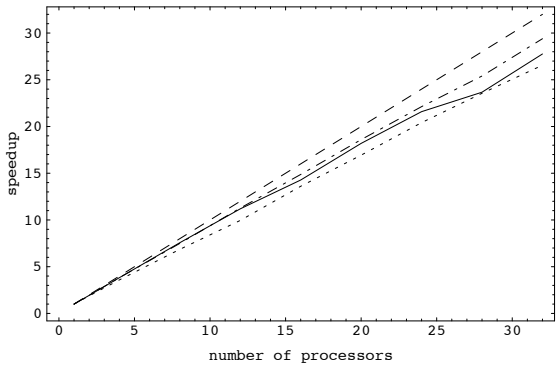


FIGURE 2.4 – Accélération de l'implémentation parallèle de Murφ

2.3.2 [NC97]

Les gains rapportés sont de 6.6 sur 8 machines et de 13.6 sur 16 machines. Un seul modèle est testé, dont l'espace d'états contient un peu plus de 150000 états. Cet outil est capable de répartir dynamiquement la charge. La figure 2.5 montre l'évolution de l'accélération en fonction de la fréquence à laquelle est effectuée l'équilibrage de charge, pour 8 et 16 machines. On voit que, que plus ce taux est élevé, moins l'accélération est bonne, ce qui indique qu'il ne faut pas chercher à équilibrer la charge trop souvent.

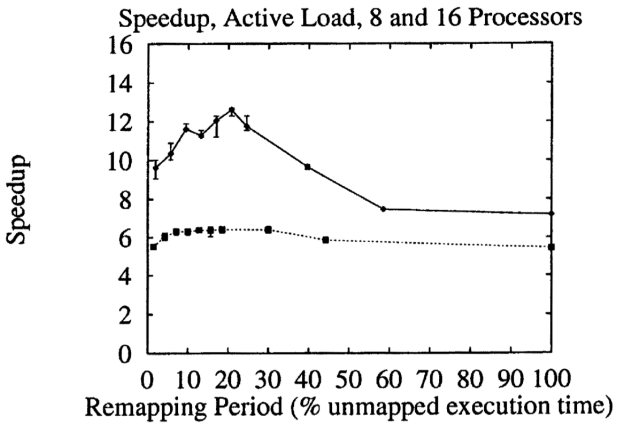


FIGURE 2.5 – Accélération de [NC97]

2.3.3 SPIN [LS99] et [HB07]

[LS99]. Il s'agit d'une implémentation répartie de SPIN. Les résultats ont été obtenus à partir de trois modèles différents, testés selon plusieurs paramètres, sur deux et quatre machines. Sur l'exemple d'un algorithme de Lamport pour l'exclu-

sion mutuelle, si le nombre d'états du système est trop faible ($< 2^9$ états), alors la version séquentielle est meilleure que celle répartie. Dans le cas où la version séquentielle sature la mémoire, alors la version répartie utilisant le moins de nœuds est la plus efficace.

Dans le cas où l'on dépasse 2^9 états, la version répartie à quatre machines est meilleure que celle à deux machines, mais l'accélération est loin d'être linéaire. Les résultats sont similaires dans le cas du modèle d'un algorithme d'élection de leader.

Les tests sont assez peu représentatifs, puisqu'ils ne sont effectués que sur quatre nœuds, ce qui ne permet pas d'évaluer le passage à l'échelle. De plus, une étude [RDBSC04] menée pour évaluer cette version répartie de SPIN a estimé que cette répartition provoque des pics d'utilisation de la mémoire trop grands.

[HB07]. Cette autre version parallèle de SPIN n'est pas répartie, mais parallèle. Aucun des résultats exhibés ne montre de gains liés parallélisme. Il se trouve que les seuls gains sont en fait ceux obtenus grâce au bon usage du compilateur.

2.3.4 CADP [GMS01]

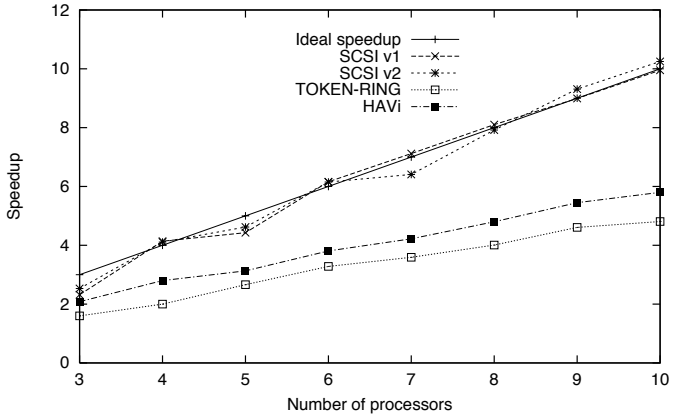


FIGURE 2.6 – Accélération de CADP

La figure 2.6 montre les accélérations obtenus par CADP, sur quatre modèles différentes. On observe des accélérations presque linéaires pour deux des quatre modèles atteignent. Par contre, pour les deux autres, elle est deux fois moins bonnes. Les auteurs expliquent ceci par le fait que les deux modèles concernés sont relativement simples à calculer, les communications devenant alors plus importantes que les calculs d'états. Ces données sont corrélées avec le fait que, lors de l'adjonction d'un *buffer* pour agréger les états avant de les envoyer, les résultats pour ces modèles sont meilleurs. En revanche, l'usage d'un *buffer* a tendance à ralentir les deux modèles nécessitant un calcul complexe.

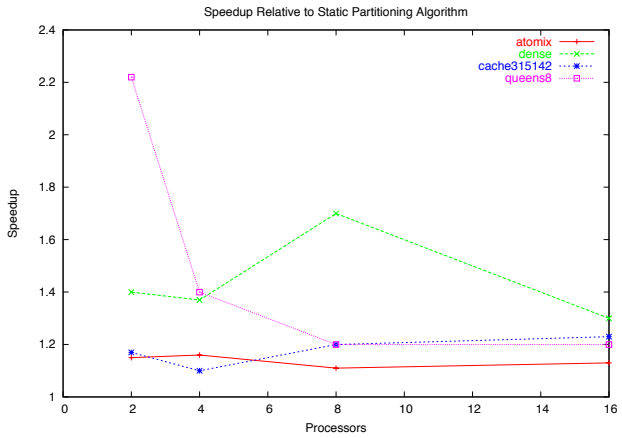


FIGURE 2.7 – Accélération de [KM05]

2.3.5 [KM05]

Les résultats présentés en figure 2.7 ont été obtenus à partir de quatre modèles différents. Comme la contribution de [KM05] concerne la répartition de charge, l'accélération présentée correspond en fait à celle que fournit leur algorithme par rapport à une version répartie avec un système de localisation statique, ce qui explique les faibles résultats. Selon les auteurs, leur technique ne fournit au pire qu'une accélération très légèrement supérieure à 1, mais elle n'engendre jamais de surcoût.

2.3.6 Discussion sur les performances de la littérature

De tous les outils rencontrés dans la littérature, seuls Murφ [SD97] et CADP [GMS01] montrent des accélérations quasi-linéaires. Si [NC97] présente presque aussi des accélérations linéaires, il faut toutefois nuancer cela par le fait qu'il faut déterminer la bonne fréquence à laquelle effectuer un ré-équilibrage de la charge. Les résultats étant présentés sur un seul modèle, il est difficile d'imaginer que cette fréquence sera la même pour chaque modèle.

Pour les résultats provenant d'outils testés sur plusieurs modèles, on voit qu'en fonction de ces modèles, l'accélération n'est pas la même, ce qui montre bien l'influence du modèle analysé sur la génération répartie.

À titre d'information, [JMB+03] propose des solutions dans le but de fournir des méthodes normalisées afin de comparer les résultats des bancs de tests de différents models checkers répartis. Un tel effort pourrait s'avérer utile à l'avenir afin de pouvoir sélectionner un *model checker* réparti. Il faudrait toutefois que ce choix puisse être fait en fonction des caractéristiques des modèles à analyser.

Il s'avère donc que malgré les efforts déployés, le *model checking* réparti et parallèle nécessite d'autres avancées, puisque rares sont les outils exhibant une

accélération linéaire. Ceci nous parait être une condition nécessaire afin de prétendre avoir un *model checker* réparti efficace. En effet, nous estimons qu'obtenir des accélérations non linéaires signifie que l'on n'a pas exploiter au mieux les ressources d'un *cluster* ou d'une architecture multi-processeurs.

2.4 Synthèse et positionnement

Pour mieux comprendre les raisons de ces demi-succès, nous avons identifiés les paramètres qui ont une influence sur la génération répartie d'un espace d'états. De plus, pour chacun d'entre eux, nous donnons le positionnement que nous avons choisi.

1. *Formalisme des modèles analysés.*

   Synthèse. À l'heure actuelle, les informations que peuvent apporter un formalisme d'entrée sont très peu utilisées. Toutes aussi rares sont les analyses statiques sur le modèle vérifié afin d'en extraire des informations utiles à la répartition.

   Positionnement. Nous considérons avant tout que la conception d'un *model checker* est une tâche difficile et qu'il faut utiliser l'existant, et de ce fait nous avons choisi de créeer une bibliothèque dévolue à la seule tâche de la parallélisation et de la répartition d'un model checker. De plus, nous intéressant pour l'instant aux seules propriétés de sûreté, nous exposons des interfaces exprimant le comportement d'un automate.

2. *Codage des états.*

   Synthèse. Nous avons vue que le codage des états doit être indépendant des données locales aux nœuds. Vis à vis de l'utilisateur, il est possible de gérer ce codage de deux manières : soit l'utilisateur est entièrement maître de son système de codage, soit il fournit des éléments qui permettront au *model checker* de forger son propre codage, à la manière de JavaPathFinder.

   Positionnement. L'approche bibliothèque suggère de ne pas imposer un codage particulier d'états, puisqu'il n'est pas envisageable d'exiger d'un *model checker* qu'il modifie ses représentations d'états. Au contraire, nous utilisons de manière transparente le codage des *model checkers* existants.

3. *Calcul des successeurs.*

Synthèse. Calculer les successeurs ne demande pas la même charge de calcul en fonction du *model checker*. Or, un calcul trop rapide peut être la raison d'un échec de la répartition. En effet, cela induit qu'il faut transférer ces états sur le réseau, celui-ci devenant un goulot d'étranglement puisqu'ayant trop d'états à transférer. Autrement dit, les calculs ne « recouvrent » pas les communications.

Positionnement. La notion de recouvrement, très importante dans les calculs répartis, n'a jamais été vraiment abordée dans la littérature. Nous souhaitons donc utiliser en priorité des *model checkers* dont le calcul des successeurs est coûteux dans l'optique de rendre maximal ce temps de calcul face au temps passé dans les communications.
De plus, nous avons fait le choix de paralléliser *et* répartir la génération, ce qui n'a jamais été effectué. Ce choix a pour but de parvenir à ce fameux recouvrement des communications, comme cela sera montré par la suite.

4. *Transfert des états.*

Synthèse. Il faut prendre en compte la possibilité qu'un état soit trop gros pour être transféré efficacement, auquel cas il faudrait peut-être mieux transférer le chemin dans le graphe d'accessibilité (en suivant les événements) amenant à l'état transmis.

Positionnement. Toujours dans l'idée de fournir une bibliothèque ayant le moins de pré-requis de la part des *model checkers*, nous ne voulons pas en exiger que les utilisateurs soient en mesure de nous décrire un chemin.

5. *Fonction de localisation des états.*

Synthèse. Il s'agit du point le plus sensible, sur lequel repose la génération répartie d'un graphe d'accessibilité. La section 2.2.2.1 explique en détail les choix possibles. Pour résumer, il faut choisir entre un système statique ou dynamique. Le premier offrant un nombre plus petit de messages échangés, au prix de l'impossibilité de changer les états de propriétaire, et réciproquement pour le système dynamique.
C'est un point qui est fortement lié à l'équilibrage de charge, car une bonne fonction de localisation devrait permettre, *a priori*, une bonne distribution de l'espace d'états, avec un minimum de messages échangés.

Positionnement. Hachage statique mis à part, les solutions proposées par la littérature sont complexes pour peu de résultats positifs. De ce fait, nous préférons privilégier la simplicité du hachage statique, qui, nous le

verrons par la suite, donne de très bons résultats.

6. *Terminaison.*

   Synthèse. Il faut concevoir la détection de terminaison de manière à ce qu'elle ne fasse pas perdre des performances. Par exemple, une politique agressive demandant trop souvent des informations pourrait saturer les nœuds avec des traitements inutiles de messages provenant du réseau.

   Positionnement. Nous avons choisi un algorithme de détection réparti de terminaison basé sur un système de compteurs [Mat87], relativement simple à mettre en place, que nous exécutons au moment opportun pour ne pas pénaliser les performances.

7. *Équilibrage de charge.*

   Synthèse. Comme le point précédent, si la politique d'équilibrage de charge s'appuie sur des demandes trop fréquentes d'information sur la charge mémoire ou processeur, les nœuds perdront en efficacité.

   Positionnement. L'équilibrage de charge est un moyen de contre-balancer les éventuels effets indésirables d'une mauvaise répartition des états sur le *cluster*, et de ce fait, nous avions décidé de la mettre en place seulement si les expériences en montrait la nécessité, ce qui ne s'est pas produit.

8. *Topologie.*

   Synthèse. Une topologie non adaptée est rapidement un frein au bon fonctionnement. Par exemple, choisir une architecture client-serveur centraliserait toutes les communications sur un nœud qui s'écroulerait très vite.

   Positionnement. Nous avons choisi d'utiliser une topologie pair-à-pair pour échanger les états, afin d'éviter qu'un nœud ne se retrouve saturé par les communications. Le graphe des liens de communications entre chaque nœud est complet : chaque nœud est connecté à tous les autres.

CHAPITRE 3

Répartition et parallélisation du calcul d'un espace d'états

Ce chapitre présente nos contributions au domaine du *model checking* parallèle et réparti. Nous nous intéressons en particulier à **la génération parallèle et répartie d'un espace d'états** pour la vérification des propriétés de sûreté. Ces travaux ont été publiés dans [**?**] et [HKTMLA07] et se sont concrétisés sous la forme d'une bibliothèque appelée LIBDMC.

Nous commençons par exposer les choix de conception par rapport à l'état de l'art 3.1. Nous montrons ensuite comment nous avons géré la répartition en section 3.2. La section 3.3 explique comment nous procédons à la vérification de propriétés de sûreté. Ensuite, la section 3.4 décrit l'architecture de la bibliothèque LIBDMC.

Étant parti du principe de développer une bibliothèque, il fallait choisir un *model checker* avec lequel l'interfacer. GreatSPN [CFGR95] a été choisi car maîtrisé, à la fois d'un point de vue logiciel et formel, au sein du laboratoire où se sont effectués ces travaux. Cette intégration est expliquée en section 3.6. Choisir GreatSPN s'est révélé riche en enseignements sur les conditions optimales permettant d'obtenir un *model checker* réparti efficace, comme le montrent les résultats des expérimentations en section 3.7.

3.1 Choix de conception

Nous avons positionné notre travail au chapitre précédent par rapport à la littérature. Le plus important de ces positionnements, car conditionnant la plupart des choix de conception, est de fournir aux *model checkers* existants une bibliothèque. En effet, cela implique de faire des hypothèses plus ou moins contraignantes pour l'existant, qu'elles soient de l'ordre technique ou formel.

Ces hypothèses sont les suivantes :

1. Le *model checker* doit-être en mesure de calculer les successeurs d'un état avec pour seule information cet état lui-même. Il ne peut donc exister de contexte global à la génération de l'espace d'états. Concrètement, cela signifie qu'il fournit une relation de franchissement qui est une fonction ne prenant que pour seul paramètre un état.
2. Chaque état doit-être entièrement indépendant des autres. Il ne peut donc partager aucune information avec d'autres états. Cette hypothèse très forte implique donc que l'on travaille exclusivement sur des espaces d'états explicites. Ceci exclut par exemple l'usage des diagrammes de décision pour stocker l'ensemble des états [1].
3. La représentation d'un état en mémoire doit être une seule zone contigüe, sans aucune indirection. Cela interdit par exemple l'usage de pointeurs. Toutefois, une procédure de sérialisation des états permet de s'affranchir de cette contrainte quelle que soit la situation.
4. Le *model checker* doit être *thread-safe*, autrement dit, il est possible d'exécuter la relation de franchissement en parallèle au sein d'un même espace mémoire.

Bien que technique, ce dernier point est très important dans la mesure où nous souhaitons non seulement répartir la génération d'un espace d'états, mais aussi la paralléliser. En effet, si nous n'exécutions qu'un seul *thread* par nœud d'un *cluster*, le temps passé dans les communications avec le reste du *cluster* ne serait pas compensé par du calcul d'états. Autrement dit, nous n'atteindrions pas un **recouvrement** des temps de communications par des temps de calcul. Ce qui nous empêche d'exploiter pleinement les ressources du *cluster*, surtout si les nœuds sont multi-processeurs.

La figure 3.1 illustre ceci pour un *thread* et la figure 3.2 pour plusieurs *threads*. Dans le premier cas, le temps passé dans les communications n'est jamais recouvert par du temps de calcul. Dans le deuxième cas, toutes les communications sont recouvertes par au moins un calcul : le nœud du *cluster* est donc en permanence en train de calculer des successeurs. Ceci est valable quelque soit le nombre de processeurs disponible sur un nœud : en effet, les communications provoquent un arrêt temporaire du *thread* de calcul. Dans le cas où il y aurait plusieurs *threads*, un autre *thread* de calcul pourra être réveillé par l'ordonnanceur. Le temps perdu par le *thread* en attente du canal de communication sera ainsi compensé par le calcul d'un autre *thread*.

Pour permettre à la LIBDMC d'intéragir avec les models checkers, nous avons défini des interfaces correspondant à des automates à l'aide de trois fonctions que doivent fournir les *model checkers* :

– une fonction fournissant l'état initial ;
– une fonction fournissant un ensemble de successeurs pour un état donné ;

1. Ce qui n'interdit pas des les utiliser pour stocker un ensemble d'états qui seraient vus comme équivalents sous une certaine relation, à la manière de [KP08]

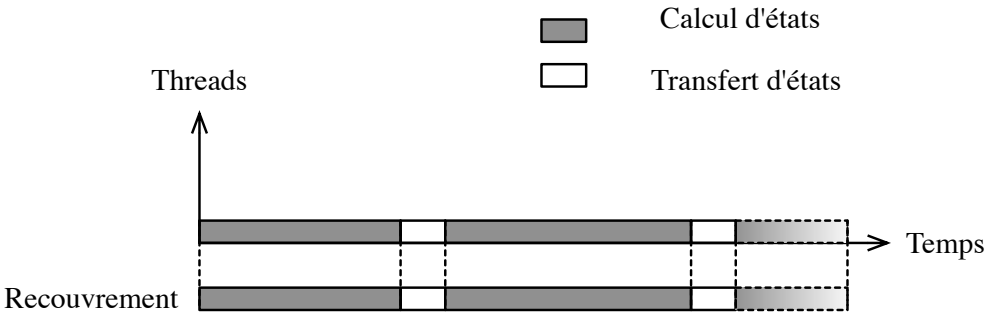


FIGURE 3.1 – Recouvrement avec un seul *thread*

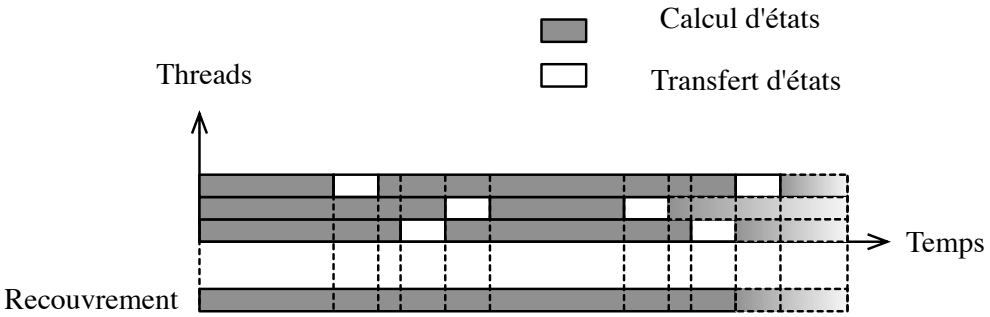


FIGURE 3.2 – Recouvrement avec plusieurs *threads*

– une fonction d'étiquetage des états par des propriétés.

D'un point de vue de Génie Logiciel, cela permet de séparer des composants dont les tâches sont clairement distinctes. D'un côté il existe des algorithmes liés à la représentation des états et de l'autre des algorithmes dédiés à la répartition.

Notons que nous prenons le parti de ne pas conserver les chemins entre états car ce sont des informations que l'on peut calculer, engendrant un gain en mémoire.

3.2 Gestion de la répartition

Topologie du réseau. La topologie pair à pair, basée sur un graphe complet, choisie pour transférer les états entre chaque nœud du *cluster* se justifie par le fait que les états sont transférés directement aux nœuds qui en sont propriétaires, sans aucun intérmédiaire. Cela limite la charge des nœuds qui serviraient de relais et celle du réseau physique.

L'inconvénient est que dans le cas d'un cluster composé de nombreuses machines, cela implique autant de connexions ouvertes sur chaque nœud, ce qui peut pénaliser les performances du système d'exploitation. Nous verrons dans les résultats présentés à la section 3.7.3 que ce point de saturation du système d'exploitation n'a pas été atteint.

Localisation des états. Nous avons choisi d'utiliser une fonction de localisation statique pour deux raisons. Tout d'abord, pour la simplicité que cela implique. Ensuite, nous avons considéré que la meilleure solution serait une fonction qui distribue de manière complètement équitable les états sur le *cluster* (même si cela implique un plus grand nombre de communications, car séparant très souvent un état de ses successeurs). Par exemple, si l'on considère un *cluster* de 10 machines sur lequel un état possède 10 successeurs, alors 9 de ces 10 successeurs seront distants et devront être communiqués aux nœuds qui en sont propriétaires.

La LIBDMC n'a aucune connaissance de la sémantique d'un modèle, puisqu'un état est représenté par un pointeur sur un emplacement mémoire et une taille en octets. La fonction de localisation doit donc pouvoir travailler sur des séquences d'octets.

Nous avons donc opté pour la somme de contrôle générée par l'algorithme MD5 afin de déterminer le propriétaire d'un état. Cette somme étant générée sur 128 bits, nous en prenons les 32 premiers, que nous voyons sous forme d'un entier. Ensuite, il suffit d'appliquer un modulo avec le nombre de nœuds du cluster pour obtenir l'identifiant du propriétaire.

Des essais ont aussi été menés avec la famille d'algorithmes SHA, mais nous avons conservé le MD5 qui est plus simple et plus rapide.

La validité de ce choix dans le cadre de la génération répartie est montrée à la section 3.7.1 qui évalue les performances de cette méthode de localisation.

Terminaison. La détection de la terminaison se fait au moyen de l'algorithme "des quatre compteurs" de Mattern [Mat87] (légèrement modifié). Il suffit , dès lors que l'on détecte un nœud inactif, de demander à chacun des nœuds le nombre de messages qu'il a reçus et envoyés, en sus de son statut (actif ou inactif). Si jamais les sommes globales (sur l'ensemble du *cluster*) des messages envoyés et reçus sont identiques et si tous les nœuds sont inactifs, alors on a détecté une terminaison. Il faut toutefois exécuter cette procédure deux fois pour éviter un faux positif, comme l'a montré Mattern.

3.3 Vérification de propriétés de sûreté

La vérification de propriétés de sûreté se fait à la volée lors de la construction de l'espace d'états. Cependant, pour que ce processus de vérification soit vraiment utile, il faut retourner une trace vers le ou les éventuels états en erreur à l'utilisateur. Dans notre cas, deux difficultés se posent :

– l'espace d'états étant partitionné sur le *cluster*, les chemins sont répartis ;
– la bibliothèque ne conserve aucune information de chemin entre les états.

Il va donc falloir reconstruire les chemins *a posteriori*. La démarche la plus simple est de partir des états en erreur et de remonter dans l'espace d'états à l'aide d'une relation de franchissement arrière. Toutefois, pour ne pas imposer trop de contraintes, nous n'exigeons pas cette relation de franchissement. De plus, pour

faciliter la compréhension de l'erreur, il faut fournir une trace de longueur minimale.

Pour obtenir cette trace de longueur minimale, il va falloir dans premier temps modifier l'algorithme de génération afin d'associer à chaque état sa distance par rapport à l'état initial. Cependant, un état calculé à une distance n en suivant un certain chemin pourra se retrouver à une distance $m < n$ en suivant un autre chemin par la suite. Il faut donc mettre à jour la distance de chacun de ses successeurs. Il suffit pour cela de réinsérer cet état en erreur dans l'ensemble des états non-visités, forçant ainsi un nouveau calcul. Nous n'avons observé aucune perte de performances malgré ce mécanisme.

La récupération d'un plus court-chemin se fait de la manière suivante : une fois qu'un état s_n en erreur est détecté, le maître l'envoie à tous les esclaves. Ceux-ci recherchent dans tous les états à la distance $distance(s_n) - 1$ ceux qui ont pour successeur s_n, en utilisant la relation de franchissement du *model checker*. Ces prédécesseurs sont renvoyés au maître qui sélectionne un état s_{n-1} parmi ceux-ci [2]. Tant que l'on ne rencontre pas un état initial dans les prédécesseurs, le processus est répété : le maître renvoie s_{n-1} à tous les esclaves. Le chemin en erreur est ensuite reconstruit en partant de l'état initial, puis en y ajoutant les états envoyés par les nœuds du *cluster*.

3.4 Architecture

Nous présentons dans cette section l'architecture de la LIBDMC. En premier lieu, nous montrons comment est gérée la génération parallèle, puis nous étendons l'architecture pour gérer la génération répartie.

3.4.1 Génération parallèle

La génération locale s'articule autour de deux composants : le *StateManager* (SM), le *FiringManager* (FM), schématisés en figure 3.3. Ils partagent une file des états en attente d'être traités.

Le *StateManager* a pour rôle de déterminer si les successeurs calculés sont de nouveaux états. Le cas échéant, il les insère dans la table d'unicité locale ainsi que dans la file d'attente des états à traiter.

Le *FiringManager* calcule les successeurs immédiats d'un état prélevé dans la file d'attente en déléguant cette opération à la relation de franchissement du *model checker*. Le parallélisme est créé en instanciant ce composant plusieurs fois, exécutés dans différents *threads*. Dès que le *FiringManager* a calculé un successeur, il le transmet au *StateManager* afin que celui-ci le traite.

2. Il est tout aussi possible de traiter tous les prédécesseurs si l'on veut avoir tous les chemins en erreur de taille minimale

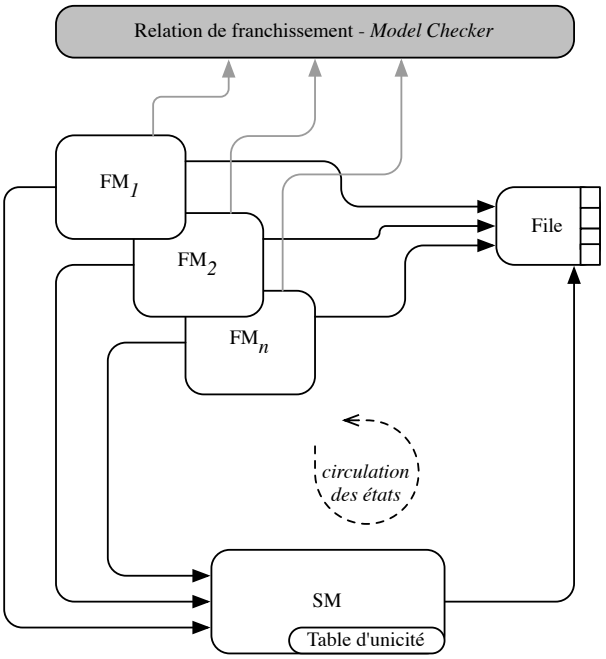


FIGURE 3.3 – Architecture - Génération parallèle

3.4.2 Génération répartie

Pour réaliser la génération répartie, nous avons besoin de deux architectures différentes : une première pour effectuer le calcul proprement dit, et une deuxième pour contrôler cette génération, c'est à dire lancer le calcul et en détecter la terminaison.

Toutes les activités réseau sont masquées par un système de mandataires : lorsqu'un nœud communique avec un autre, il le fait via un objet répondant aux même interfaces que le service distant invoqué. Cela a plusieurs avantages : un composant peut-être local ou distant sans que le composant l'appelant en ait connaissance, simplifiant ainsi grandement le travail de programmation puisqu'un composant n'a pas à être réécrit selon qu'il invoque un service distant ou local. Cela permet également de migrer dynamiquement un composant vers un autre nœud moins chargé. Bien que nous n'utilisions pas cette possibilité, cela pourra s'avérer utile pour mettre en place un mécanisme d'équilibrage de charge.

Architecture pour la génération répartie. La figure 3.4 montre la partie de l'architecture de la bibliothèque concernant la génération répartie de l'espace d'états. Elle est très semblable à celle de la génération locale , à ceci près qu'il existe trois nouveaux composants : *StateManager Proxy*, *StateManager Service*, et *DistributedStateManager*.

Le *DistributedStateManager* est maintenant l'entité à laquelle s'adresse les

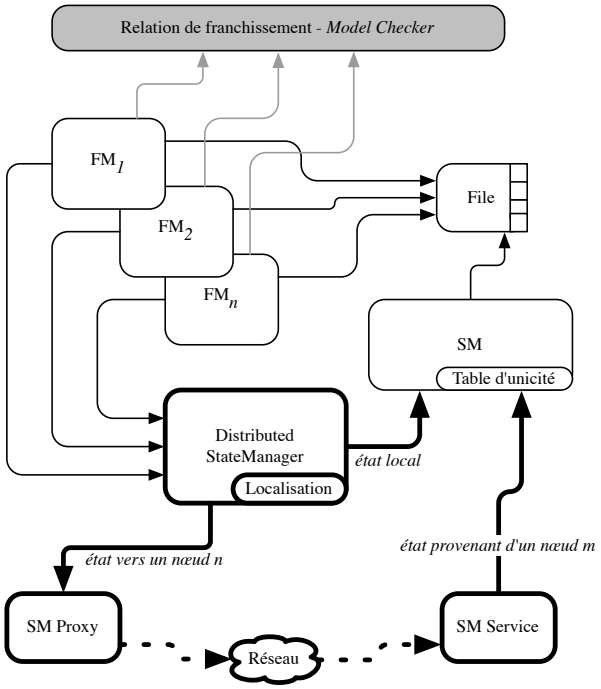


FIGURE 3.4 – Architecture - Génération répartie

threads de génération. Le composant possède un tableau, de taille égale au nombre de nœuds, qui stocke des objets représentant chaque nœud réel du *cluster*. Chaque entrée de ce tableau correspond donc à la vue qu'a un nœud des autres. Ce composant sert donc « d'aiguilleur » 'aux états : il est en charge du calcul de la localisation d'un état pour ensuite en déléguer le traitement à l'objet mandataire (*StateManager Proxy*) du nœud propriétaire. Dans le cas d'un état possédé par un nœud distant, il sera transmis directement à son propriétaire (*StateManager Service*). Dans le cas d'un état local, il sera directement transmis au *StateManager*.

Architecture pour le contrôle de la génération. Même si la génération répartie adopte une topologie pair à pair, il est nécessaire d'avoir un nœud maître qui déclenche cette génération et en détecte la terminaison. On se retrouve donc dans ce cas avec une topologie centralisée. La figure 3.5 présente l'architecture concernant le contrôle de la génération.

Le nœud maître est un nœud de calcul comme tous les autres sauf qu'il est en mesure d'ordonner à chacun d'entre eux de démarrer la génération ou de s'arrêter. Il voit chaque nœud de calcul comme un FiringManager. Il possède donc un tableau de mandataires (FiringManagerProxy) qui représente chacun des nœuds distants. Les ordres sont donnés via ces mandataires.

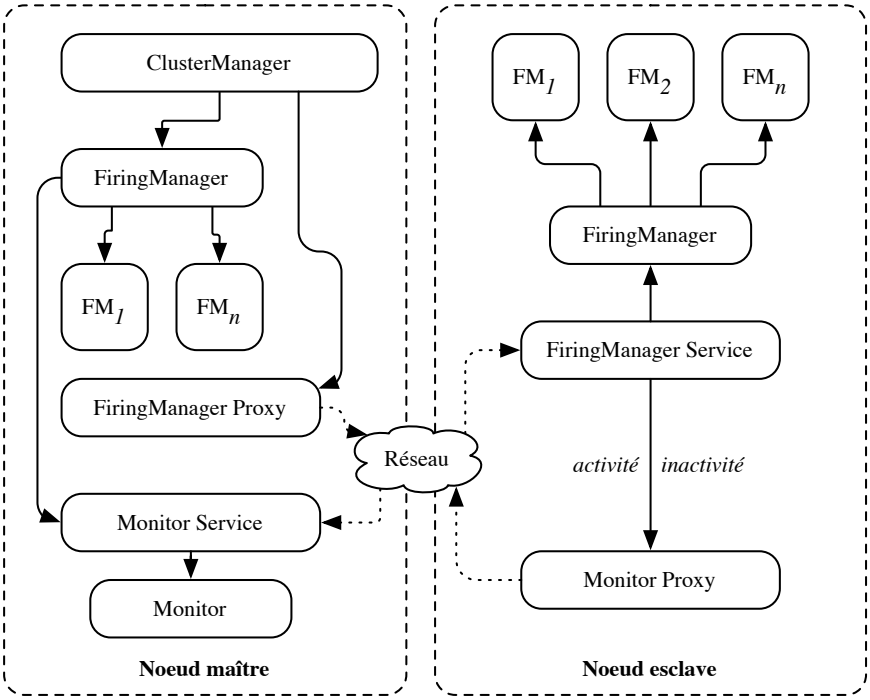


FIGURE 3.5 – Architecture - Relations maître-esclaves

De plus, il surveille l'activité des esclaves pour déclencher un processus de détection de terminaison.

Cette architecture client-serveur n'est pas pénalisante pour les performances car elle engendre très peu de communications. Celles-ci ne sont créées que lorsque des ordres sont transmis ou lors de la détection de terminaison.

3.5 La bibliothèque LIBDMC

La LIBDMC est une bibliothèque qui implémente l'architecture décrite dans les section précédentes. Elle est écrite en C++, permettant ainsi l'intégration de *model checkers* écrits en C ou C++, ce qui est très généralement le cas pour des raisons d'efficacité.

Les communications sur le réseau sont effectuées à l'aide des mécanismes standards UNIX que sont les *sockets TCP*. Il fut dans un premier temps envisagé d'utiliser MPI, mais à l'époque de l'implémentation de la LIBDMC, la version 1.1 de cette norme n'était pas suffisamment *thread-safe*, ce qui est totalement rédhibitoire pour notre cas d'utilisation. La version 2 de MPI apporte cet aspect *thread-safe*, mais aucune implémentation n'était suffisamment complète au moment de l'écriture de notre bibliothèque. Cela nous avantage car la LIBDMC dépend ainsi de moins de composants externes.

```
#include "dmc.hh"
#include "simple_model/simple_model.hh"
#include "simple_model/simple_model_factory.hh"

int main()
{
        simple_model::simple_model_factory smf();
        dmc::dmc_manager dmc(smf);
        dmc.start_generation();

        return 0;
}
```

FIGURE 3.6 – Lancement d'une génération d'un espace d'états avec la LIBDMC

L'algorithme MD5 que nous utilisons provient de la bibliothèque OpenSSL, relativement standard sur les systèmes d'exploitation modernes.

Les threads sont ceux standardisés par la norme POSIX [Ope] (les "*pthreads*"), donc portables. De plus, nous avons renoncé à utiliser des appels système qui, malgré leur présences dans la norme POSIX, ne se comportent pas de la même manière sur tous les systèmes.

La figure 3.6 illustre la simplicité de l'usage de la bibliothèque : l'utilisateur fournit une fabrique permettant d'instancier ses composants décrivant la relation de franchissement, puis la fournit à la bibliothèque avant de lui demander de démarrer la génération. Un fichier de configuration lu par la bibliothèque lui permettra de savoir s'il faut effectuer une génération répartie ou seulement parallèle.

L'annexe A expose les interfaces de la LIBDMC.

3.6 Intégration avec GreatSPN

GreatSPN [CFGR95] est une plate-forme de *model checking* développée par l'Université de Turin. Nous en utilisons un composant de génération d'espace d'états symboliques selon l'approche de [CDFH93]. Par abus de language, nous nommerons aussi ce composant GreatSPN. Ces travaux se basent sur les informations de symétries que peuvent fournir les réseaux de Petri "bien formés" [3]. L'espace d'états est quotienté par les relations d'équivalence que donnent ces symétries.

Cette procédure de compression nécessite que chaque état du graphe d'états quotienté nouvellement calculé passe par une coûteuse procédure de canonisation. Ce dernier point a toute son importance car nous considérons qu'il est essentiel que le calcul des successeurs soit conséquent pour favoriser le recouvre-

3. Maintenant appelés « réseaux de Petri symétriques »

ment des communications. La discussion sur les résultats de la LIBDMC couplée à GreatSPN reviendra sur ce point en section 3.7.

La première étape dans l'intégration de GreatSPN a été d'identifier les fonctions qui calculent les successeurs d'un état. Ce travail a été simplifié grâce à un travail précédent similaire dont le but était d'ajouter des capacités de vérification de formules LTL à GreatSPN en utilisant la bibliothèque Spot [DLP04].

Le fait de déléguer entièrement le calcul des successeurs à GreatSPN a permis de n'avoir aucun algorithme à (re-)développer. Les algorithmes existants liés à la représentation des états sont clairement séparés de ceux liés à la répartition. Remarquons qu'il n'était pas envisageable de développer à nouveau ces algorithmes car ils sont très complexes et font partie d'une architecture vieille de plus de 10 ans. Le succès de l'intégration de GreatSPN avec la LIBDMC est une preuve du bien fondé de l'approche bibliothèque que nous avons adoptée : cette intégration s'est faite en moins d'une journée par une seule personne.

Il y eut toutefois une difficulté d'ordre technique : nous exigeons des *model checkers* qu'ils soient *thread-safe*. Cependant, le code relativement ancien (écrit en C) de GreatSPN utilise des variables globales le rendant inutilisable dans un environnement parallèle. Pour contourner ce problème, nous avons encapsulé les fonctions liées à la génération d'états de GreatSPN dans une bibliothèque dynamique, qui est chargée une fois par *thread* à partir de fichier différents. Les bibliothèques dynamiques étant chargées dans des espaces distincts de la mémoire, nous avons pu ainsi rendre GreatSPN *thread-safe*.

3.7 Expérimentations

Nous avons validé nos choix à travers des expérimentations menées sur un *cluster* composé de 22 machines bi-processeurs Xeon 64 bits doté de la technologie *Hyper-threading*[4], cadencés à 2,8 GHz et possédant 2 Go de RAM chacun. Le réseau de communication est en Gigabit ethernet. Des expérimentations supplémentaires pour tester le passage à l'échelle ont été menées sur le Grid'5000 [CCD+05].

Les tests se sont portés sur la génération répartie et parallèle de trois modèles à l'aide de GreatSPN dont l'intégration à la LIBDMC a été décrite en section 3.6 :

- le diner de philosophes. Ce modèle est paramétré en fonction du nombre de philosophes. Les valeurs utilisées varient de 9 à 15 ;
- un modèle de producteurs-consommateurs. Le paramètre est le nombre de producteurs et de consommateurs, variant entre 100 et 400 ;
- la modélisation du noyau de l'intergiciel PolyORB. Le paramètre est le nombre de *threads* modélisés dans le noyau. Les valeurs utilisées sont : 4, 11, 17, 20 et 25.

Pour indiquer un paramétrage particulier d'un modèle, nous ferons suivre son nom par la taille du paramètre (i.e. "PolyORB(17)").

4. http://www.intel.com/technology/platform-technology/hyper-threading

Cette validation passe par deux étapes. La première est de vérifier que la fonction de localisation des états répartit équitablement les états sur le *cluster* et qu'elle n'engendre pas trop de communications (section 3.7.1). La deuxième étape est de mesurer les performances en calculant les accélérations, donc de comparer les vitesses d'exécution parallèle (section 3.7.2) et répartie (section 3.7.3) à la vitesse d'exécution en séquentiel.

3.7.1 Localisation des états

Nous nous intéressons ici à déterminer dans quelle mesure la fonction de hachage statique est efficace. Premièrement, nous mesurons donc comment sont distribués les états sur le *cluster*, afin de s'assurer de l'homogénéité de la répartition. Ensuite, nous mesurons le nombre de communications engendrées par cette fonction de hachage. Enfin, nous mesurons l'activité des nœuds afin de déterminer s'ils sont tous actifs en permanence.

Répartition des états. Il est important de vérifier si la localisation calculée à partir du MD5 répartit les états de manière uniforme sur les nœuds. En effet, cela indique que chaque nœud se verra attribuer une quantité égale d'états à stocker, et donc qu'aucun nœud ne saturera sa mémoire. Nous avons donc mesuré pour l'ensemble des modèles testés le nombre d'états possédés par chaque nœud.

Ces mesures sont présentées dans les tableaux 3.1, 3.2 et 3.3 pour les modèles respectifs Philosophes, Prodcteur-Consommateur et PolyORB. Pour chacun de ces modèles, nous avons fait varier le nombre d'états et de nœuds. Nous avons mesuré le nombre d'états possédés par chaque nœud, puis calculé le nombre idéal d'états qu'aurait dû posséder chaque nœud (colonne « Moyenne »). Enfin, nous avons calculé l'écart-type afin de déterminer quelle était la dispersion autour de la moyenne. Nous avons aussi exprimé l'écart-type sous forme de pourcentage par rapport à la moyenne afin de mieux visualiser le rapport entre ces quantités.

Philosophes	Nb nœuds	Nb états	Moyenne	Écart-type	%
9	4	10 257	2 564	90	3,5%
9	22	10 257	466	25	5,4%
12	4	347 337	86 834	427	0,5%
12	22	347 337	15 788	689	4.4%
15	4	12 545 925	3 136 481	792	<0,1%
15	22	12 545 925	570 269	24 179	4,2%

TABLE 3.1 – Répartition des états pour le modèle Philosophes

La répartition est très homogène : l'écart type est petit devant la moyenne, et ce, pour de grands ou petits espaces d'états. Cela nous assure qu'aucun nœud n'est surchargé en mémoire par rapport à un autre. On remarque également que pour

Prod. et Cons.	Nb nœuds	Nb états	Moyenne	Écart-type	%
100	4	176 851	44 213	202	0,5%
100	20	176 851	8 8843	316	3,57%
400	4	10 827 401	2 706 850	1327	<0,1%
400	20	10 827 401	541 370	17 438	3,22 %

TABLE 3.2 – Répartition des états pour le modèle Producteurs-Consommateurs

Threads	Nb nœuds	Nb états	Moyenne	Écart-type	%
4	4	171 444	42 861	179	0,5%
4	20	171 444	8572	289	3,4%
11	4	3 366 471	841 618	1021	0,1%
11	20	3 366 471	168 324	5393	3,2%
17	16	12 055 899	752 494	1131	0,2%
17	20	12 055 899	602 795	19 557	3,2%
25	20	37 623 267	1 881 163	60 540	3,2%

TABLE 3.3 – Répartition des états pour le modèle PolyORB

un nombre de nœuds multiple de 2, la répartition devient encore plus homogène, cela étant certainement à attribuer au MD5.

Nombre d'états transmis. Une répartition homogène n'assure pas nécessairement un nombre réduit de communications. Nous avons donc mesuré le nombre d'états effectivement envoyés sur le réseau pour chaque modèle. La figure 3.7 montre la proportion d'états envoyés sur le réseau par rapport à ceux calculés. Nous n'avons pas reproduit ici ces courbes pour plusieurs paramètres pour chaque modèle, puisqu'elles sont à chaque fois très similaires.

On voit que le nombre d'états envoyés est extrêmement grand. En fait, dès que l'on utilise plus de 8 nœuds, plus de 9 états sur 10 n'appartiennent pas au nœud qui l'a calculé.

Cependant, au vu de l'accélération obtenue (section 3.7.3), il est clair que cela ne pose aucun problème. Il est vrai que les tests ont été menés sur un réseau en Gigabit ethernet, qui offre une bande-passante de 125 Mo/s. Il se pourrait donc que ce soit la raison pour laquelle le nombre important d'états transmis n'est pas un facteur limitant.

Pour vérifier cela, nous avons donc mesuré l'utilisation du réseau. La figure 3.8 montre pour le modèle PolyORB(25) (avec une configuration du *cluster* utilisant 20 nœuds) la bande-passante de chaque nœud à l'échelle de la capacité du réseau Gigabit. On voit que notre usage de la bande-passante est très faible. Pour mieux visualiser cette bande-passante, nous la présentons à une échelle réduite à la figure 3.9 : la moyenne est de l'ordre de 230 Ko/s par nœud, ce qui fait une charge

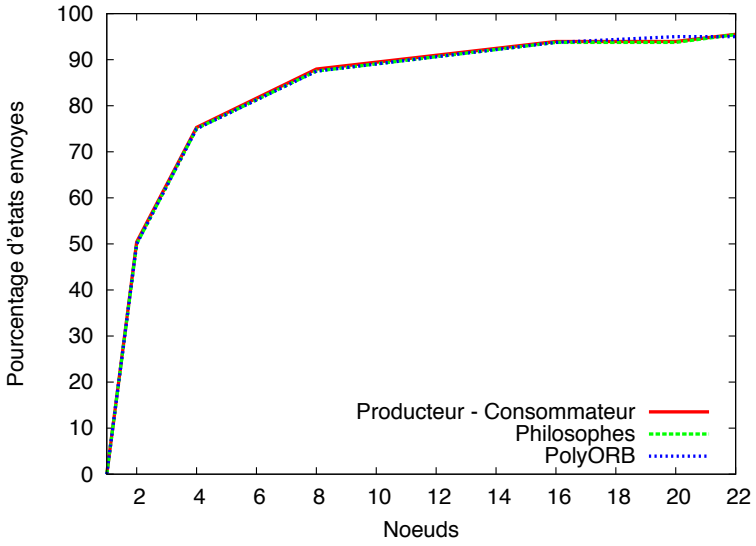


FIGURE 3.7 – Pourcentage d'états envoyés en fonction du nombre de nœuds utilisés sur le *cluster*

totale de 4,6 Mo/s. Les mesures relevées pour les autres modèles ont révélé un usage total d'au maximum 30 Mo/s, charge que n'importe quel commutateur réseau moderne d'un *cluster* supporte aisément. La bande-passante n'est donc pas une limitation à notre *model checker* réparti.

Activité des nœuds. La taille de la file des nouveaux états en attente d'être traités est un bon indicateur d'activité des nœuds. En effet, tant qu'elle n'est pas vide, les *threads* de génération sont soit en train traiter un état, soit en train d'émettre un état sur le réseau. De plus, comparer la taille des files d'attentes de chaque nœud à intervalle de temps régulier permet de déterminer si la charge de calcul est bien répartie ou non.

Nous avons donc mesuré toutes les secondes la taille de la file de chaque nœud pour les modèles Philosophes(14) (figures 3.10 et 3.11), PolyORB(11) (figures 3.12, 3.13) et PolyORB(25) (3.14).

Dans le cas de Philosophes(14), on observe à la figure 3.10 que les tailles sont relativement les mêmes au cours du temps avec un *cluster* de 16 nœuds. On observe le même comportement pour un cluster avec 22 nœuds à la figure 3.11, à ceci près qu'il apparaît deux groupes de courbes. Cela signifie dans le cas de la première figure que les charges de travail sont homogènes. Pour le deuxième cas, cela implique non pas que des nœuds sont inactifs pendant la génération, mais seulement que certains finissent leurs tâches plus tôt que d'autres. Nous corrélons cette observation qu'il existe deux groupes à celle de la distribution des états où dès lors que l'on utilise un nombre de nœuds différend de 2, alors la répartition

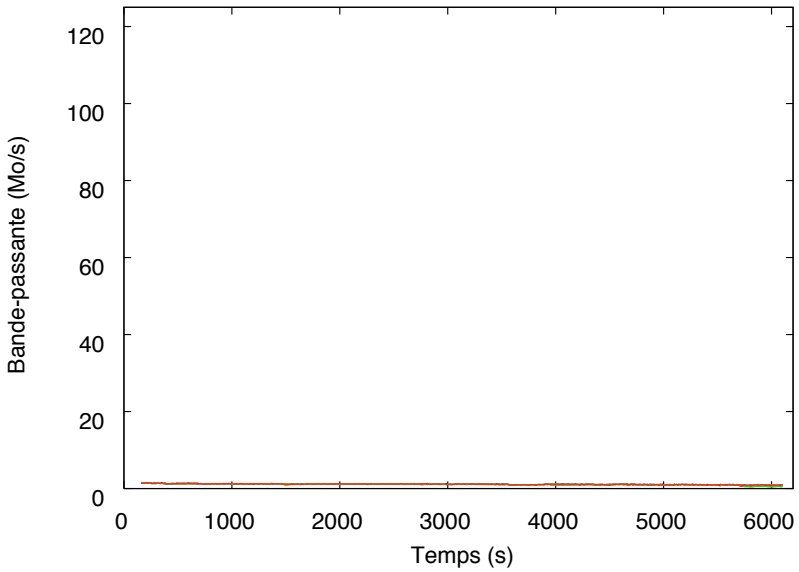


FIGURE 3.8 – Bande-passante de chaque nœud pour PolyORB (1)

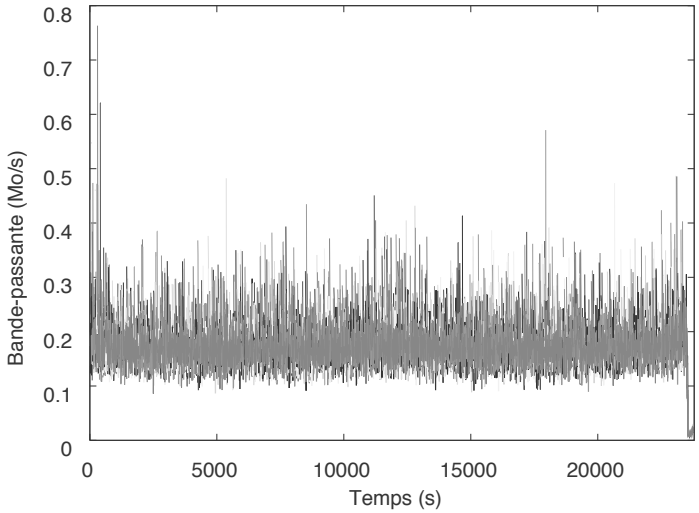


FIGURE 3.9 – Bande-passante de chaque nœud pour PolyORB (2)

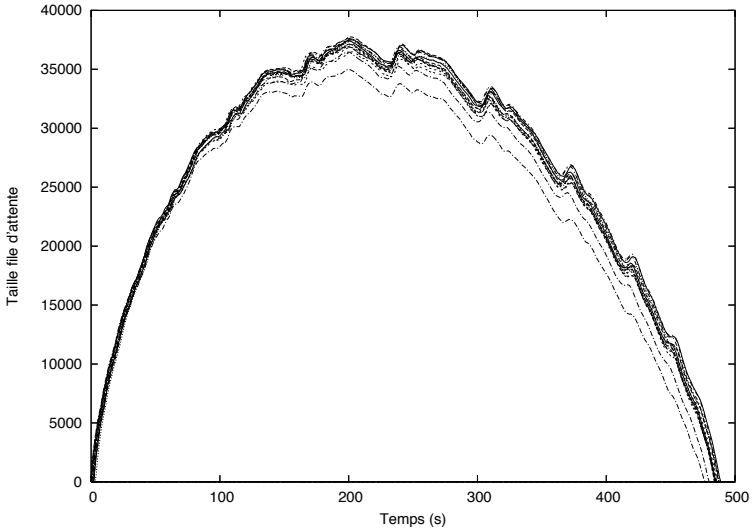


FIGURE 3.10 – Taille des files d'états à explorer en fonction du temps - Philosophes avec 14 philosophes et 16 nœuds

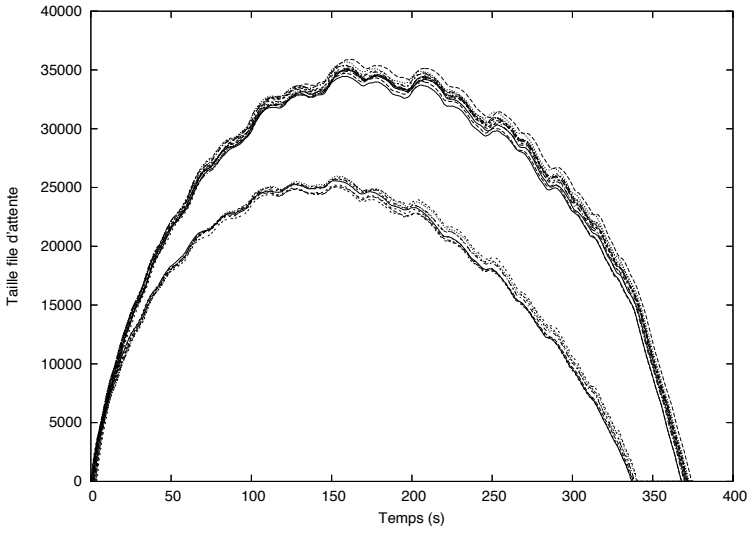


FIGURE 3.11 – Taille des files d'états à explorer en fonction du temps - Philosophes avec 14 philosophes et 22 nœuds

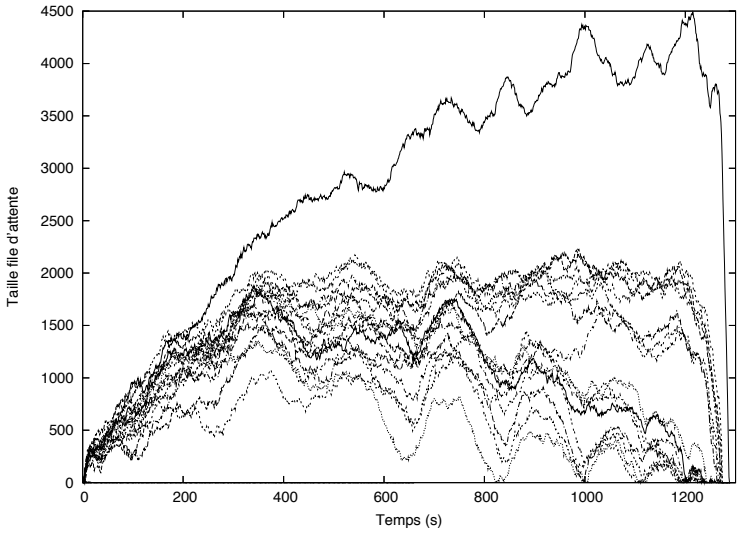


FIGURE 3.12 – Taille des files d'états à explorer en fonction du temps - PolyORB avec 11 *threads* modélisés et 16 nœuds

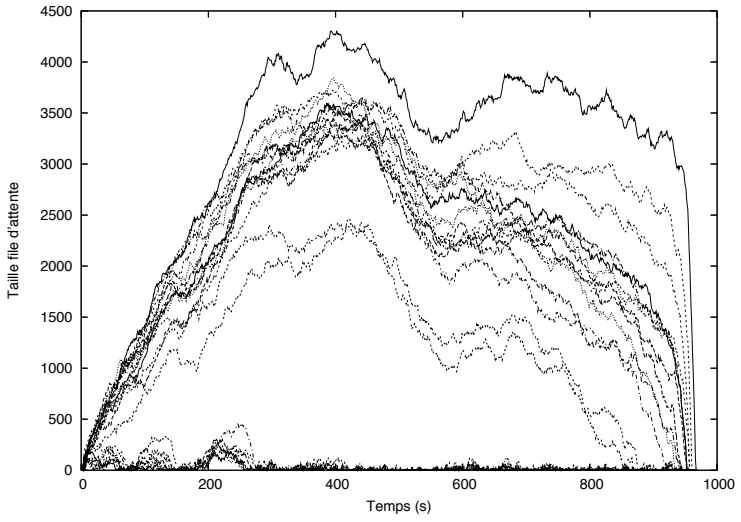


FIGURE 3.13 – Taille des files d'états à explorer en fonction du temps - PolyORB avec 11 *threads* modélisés et 22 nœuds

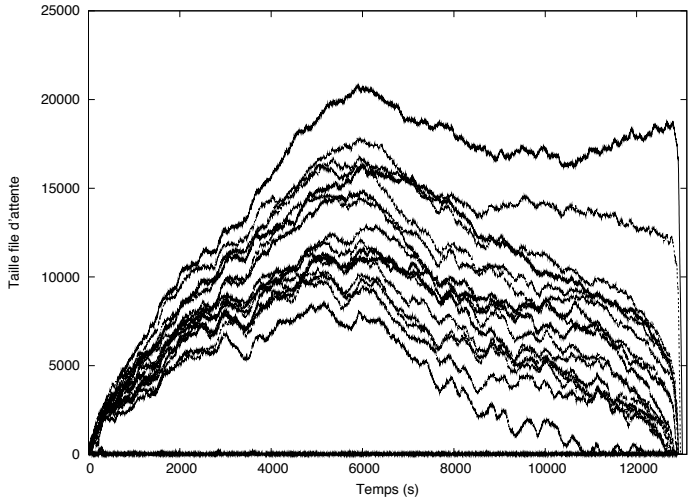


FIGURE 3.14 – Taille des files d'états à explorer en fonction du temps - PolyORB avec 25 *threads* modélisés et 20 nœuds

est moins homogène. Quelle qu'en soit la raison, ce découpage en deux groupes n'est pas un problème majeur car tous les nœuds ont en permanence du travail à effectuer.

Pour le modèle PolyORB(11), on observe une disparité flagrante dans les tailles des files, que ce soit pour un cluster avec 16 ou 22 nœuds. Il se pourrait que cela soit dû à une mauvaise distribution de l'espace d'états. Autrement dit, (même si l'on a observé en 3.7.1 une bonne localisation des états) à un instant donné pendant la génération, la probabilité pour que les successeurs d'un état soient répartis équitablement est plus faible pour ce modèle que pour celui des philosophes. Dans le cas de la figure 3.13 on observe même que certains nœuds sont très largement en dessous des autres en terme de charge de travail. Une autre explication de ce problème pourrait être le manque de régularité du modèle : une partie des états serait plus coûteuse à calculer, amenant à une mauvaise répartition de la charge de travail.

Par contre, la figure 3.14, représentant la taille des files pour PolyORB(25), montre que les tailles de files tendent à s'homogénéiser. Cela s'explique par le fait que plus un modèle devient gros, plus il y a de travail à effectuer pour chaque nœud.

3.7.2 Accélérations en génération parallèle

Nous présentons ici les résultats obtenus lors de la parallélisation de GreatSPN. Il n'est donc fait pour l'instant usage que d'une seule machine bi-processeurs, les communications étant désactivées car inutiles.

Les accélérations ont été calculées par rapport à l'exécution séquentielle de GreatSPN sur un nœud du *cluster*. Les résultats sont présentés à la figure 3.15 pour les philosophes et au tableau 3.4 pour PolyORB(4) et PolyORB(11). En raison de trop long temps de calculs, ce dernier n'a été testé que sur 4 *threads* de génération.

Threads modélisés	4	11
Accélération	2,09	2,13

TABLE 3.4 – Accélérations en génération locale pour le modèle PolyORB

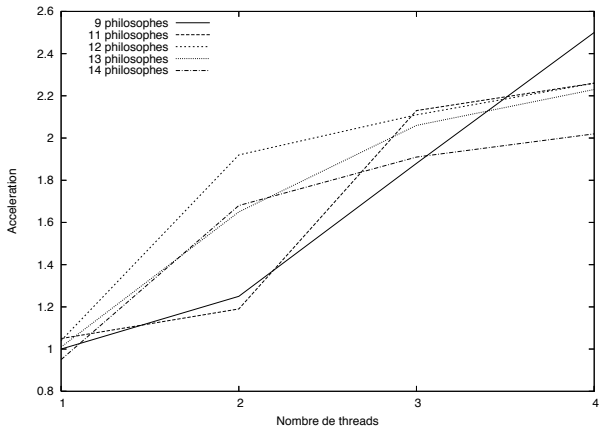


FIGURE 3.15 – Accélérations en génération locale pour le modèle des Philosophes, pour 9,11,12,13 et 14 philosophes

La première observation est que l'on obtient des accélérations **supra-linéaires**. En effet, on peut aller jusqu'à 2,6 fois plus vite, alors qu'il n'y a que deux processeurs sur les machines du *cluster*. Cette observation surprenante au premier abord n'est en réalité qu'une conséquence de la technologie *Hyper-threading* d'Intel. Cette dernière a pour but de favoriser l'exécution des applications parallèles, et nous profitons pleinement de cette technologie. Lors d'essais sur une machine multi-processeurs PowerPC 970, qui ne sont pas doté d'une technologie similaire, l'accélération n'est pas supra-linéaire, ce qui conforte cette hypothèse. Cette dernière est aussi confirmée par [KM03].

On peut ensuite observer que plus le modèle est gros, plus l'accélération est grande. Ceci s'explique par le fait que plus il y a de charge de travail, moins il a de risque qu'un processeur se retrouve oisif.

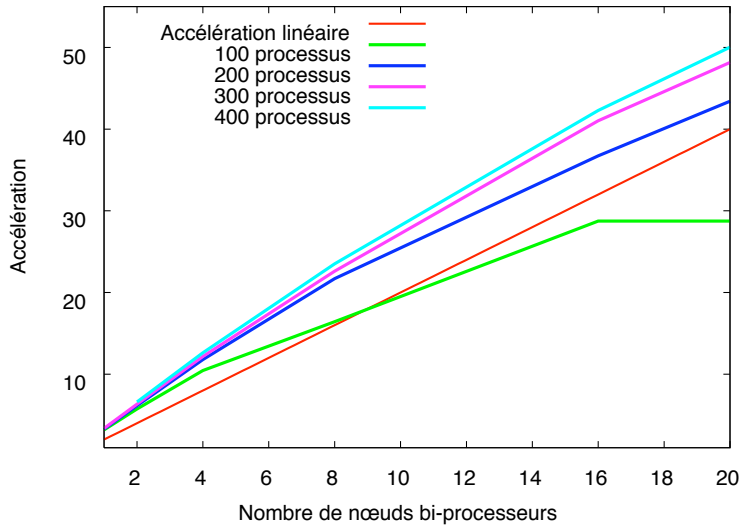


FIGURE 3.16 – Accélération en génération répartie pour le modèle Producteur-Consommateur

3.7.3 Accélérations en génération répartie

Cette section présente les accélérations obtenues dans le cadre d'une génération répartie. Chaque nœud du *cluster* utilise 4 *threads* de génération. Le temps séquentiel de référence est encore celui de la version originale de GreatSPN. Comme les nœuds du *cluster* possèdent deux processeurs chacun, nous considérons que l'accélération à atteindre n'est pas de n si l'on utilise n machines, mais $2 \times n$ car il y a en réalité deux fois plus d'unités de calcul qu'il n'y a de nœuds.

La figure 3.16 présente les résultats obtenus pour le modèle Producteur-Consommateur. On voit que les accélérations sont toujours plus que linéaires, pour atteindre jusqu'à une accélération de 50 sur 20 machines bi-processeurs. Tout comme la génération parallèle de la section précédente, cela s'attribue à la technologie d'Intel. Notons que le modèle le plus petit (pour 100 producteurs et consommateurs) ne se génère pas plus vite dès lors que l'on utilise plus de 16 nœuds. Cela est dû au fait que ce modèle devient "trop facile" à calculer et donc que des machines se retrouvent oisives.

La figure 3.17 montre le temps nécessaire pour générer l'espace d'états de ce même modèle, en fonction du nombre de processus modélisés : on passe de 2 heures et 20 minutes en génération séquentielle à moins de trois minutes sur le *cluster*.

La figure 3.18 montre les accélérations pour le modèle du diner des philosophes. Dans cette figure, l'accélération pour Philosophes(15) n'est pas représen-

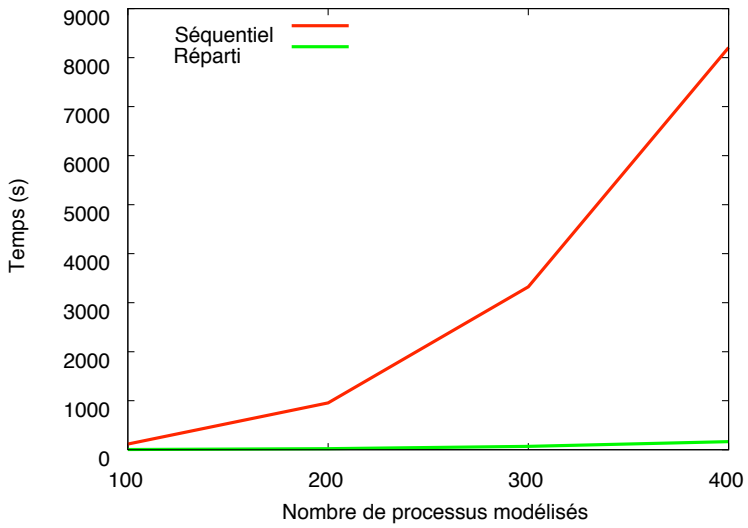


FIGURE 3.17 – Temps de génération en fonction du nombre de processus modélisés pour le modèle Producteur-Consommateur

tée puisque la version séquentielle n'a pas pu générer l'espace d'états correspondant. Pour ce modèle, il a fallu au minimum 4 nœuds du *cluster* afin de pouvoir contenir en mémoire tout l'espace d'états. La figure 3.19 montre les temps d'exécution en génération répartie et séquentielle.

On observe encore une excellente accélération allant jusqu'à 44, à l'exception de Philosophes(9) qui plafonne au-delà de 16 nœuds. La raison en étant encore le fait que le modèle est trop simple à calculer pour cette taille, et que donc des nœuds se retrouvent oisifs.

Enfin, la figure 3.20 montre les accélérations obtenues pour PolyORB(4,11,17), et la figure 3.21 les temps d'exécutions pour les générations réparties et séquentielles. Les accélérations n'ont pu être calculées au-delà de 17 *threads* modélisés car le temps de génération sur une seule machine est trop long. Le temps nécessaire pour générer l'espace d'états de ce modèle est de plus de 46 heures, alors que la génération répartie se termine en 1 heure.

Nous avons ensuite mesuré quel était l'impact de l'utilisation de plusieurs *threads* sur un même nœud du *cluster*. La figure 3.22 montre les accélérations obtenues lors de la génération de l'espace d'états du modèle Philosophes(14). En imaginant dans un premier temps que les nœuds sont dotés d'un seul processeur, le passage d'un à deux *threads* de générations permet d'obtenir une meilleure accélération. Ensuite, l'utilisation d'un ou deux *threads* supplémentaires permet d'exploiter pleinement les ressources du *cluster*. L'utilisation de plusieurs *threads*

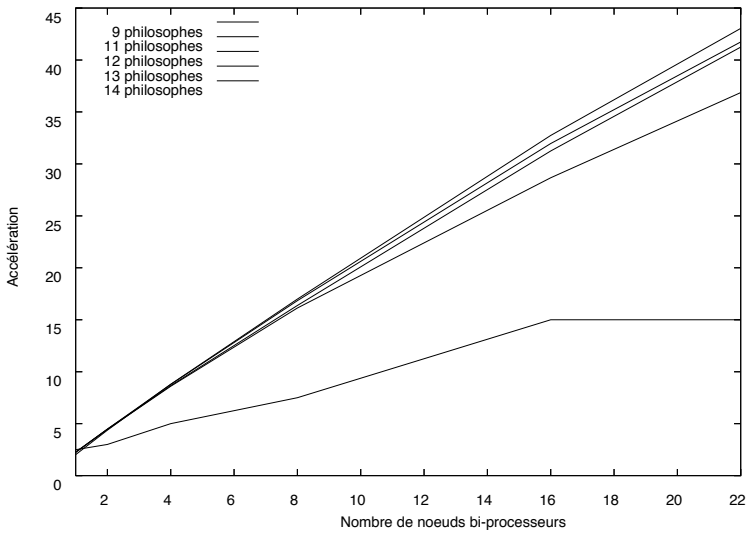


FIGURE 3.18 – Accélération en génération répartie pour le modèle des Philosophes, pour 9, 11, 12, 13 et 14 philosophes

permet donc de profiter au mieux des ressources de calcul disponibles.

Nous mesurons ensuite le passage à l'échelle : la figure 3.23 montre les accélérations pour la génération de de Philosophes(13) et Philosophes(15). Ces mesures ont été effectuées sur la grille de calcul Grid'5000 [CCD+05] car le *cluster* que nous avons utilisé pour les expérimentations précédentes ne possède pas assez de nœuds.

On observe qu'au delà de 32 nœuds, l'accélération idéale n'est plus atteinte. Cependant, l'accélération obtenue pour Philosophes(15) est meilleure que pour Philosophes(13). Nous attribuons encore ce phénomène au fait que les modèles deviennent trop simples pour une telle puissance de calcul. En réalité cela montre juste que l'utilisation d'un trop grand nombre de ressources de calcul n'apporte aucun gain réel. Par exemple, la génération du modèle Philosophes(15) prend 699 secondes en utilisant 32 nœuds et 524 secondes avec 50 nœuds.

Toutefois, le temps de calcul d'un espace d'états étant *a priori* inconnu, il n'est pas possible de connaître à l'avance le nombre de ressources nécessaires.

3.7.4 Discussion sur les résultats

Les accélérations obtenues sont toujours au moins linéaires, sauf dans le cas de modèles trop simples. On a ainsi des accélérations atteignant 50, ce qui est largement supérieure à l'accélération idéale pour 20 machines dotées de 2 proces-

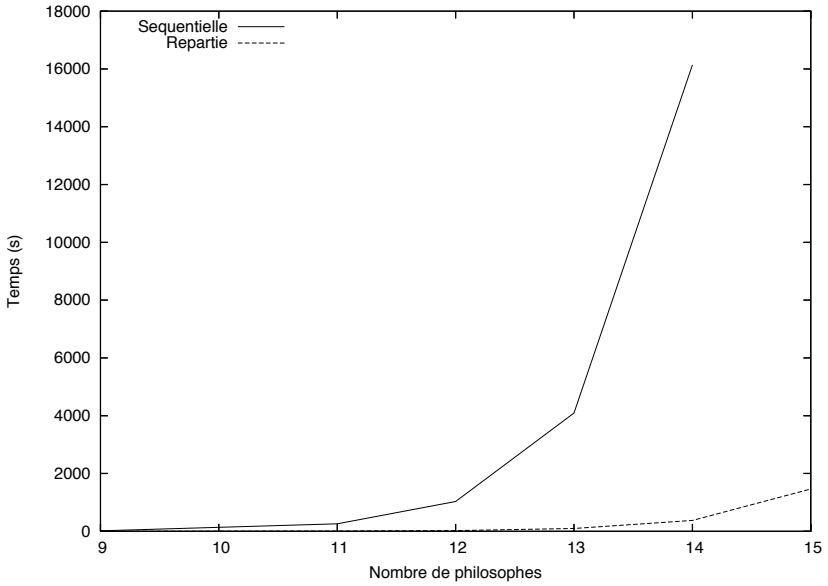


FIGURE 3.19 – Temps d'exécution en génération répartie et séquentielle pour le modèle des philosophes

seurs chacune. Cela est dû au fait que les processeurs sont munis de la technologie *Hyper-Threading*, qui améliore le parallélisme entre les *threads*.

La génération répartie a permis de repousser deux problèmes : le problème de l'explosion combinatoire et celui du temps d'exécution. En effet, on a vu qu'il est possible de générer l'espace d'états pour 15 philosophes, ce qui n'était pas possible en génération séquentielle. De plus, pour le modèle PolyORB, le facteur limitant qu'était le temps nécessaire à la génération séquentielle a pu être réduit : le calcul pour 17 *threads* modélisés est passé de plus de 46 heures à un peu plus d'une heure. Il a été possible de générer l'espace d'états pour 20 et 25 *threads* modélisés, ce qui n'avait pas été tenté auparavant.

Pour continuer sur l'exemple de PolyORB, le mauvais résultat obtenu en 3.7.1 est à commenter à la lumière de celui que l'on observe à la figure 3.20 : même si la répartition de la charge est loin d'être optimale, on observe une excellente accélération. Cela signifie que l'on peut encore gagner en vitesse puisque des nœuds étaient faiblement chargés. Un équilibrage de charge permettrait d'obtenir des performances encore meilleures.

On observe que plus l'espace d'états à générer est grand, plus l'accélération est grande. Ceci s'explique par le fait qu'avec la taille grandit la probabilité qu'un état ait plus de successeurs immédiats. De ce fait, la localisation basée sur le MD5 étant homogène, il y a de fortes chances que chacun de ces successeurs soit possédé par un nœud distant, impliquant ainsi une meilleure répartition de la charge. Nous avons vu aussi que notre usage en bande-passante reste faible même si en

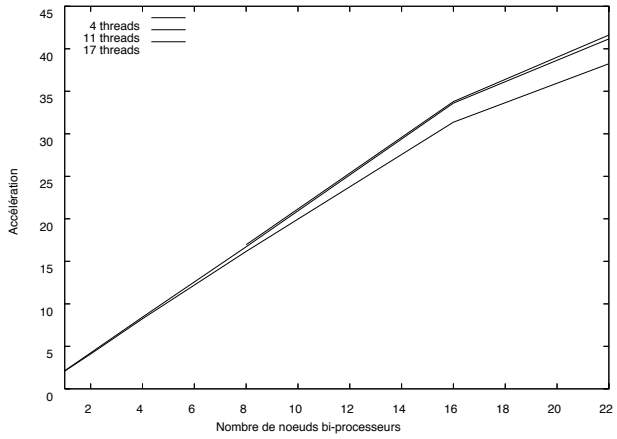


FIGURE 3.20 – Accélération en génération répartie pour le modèle PolyORB

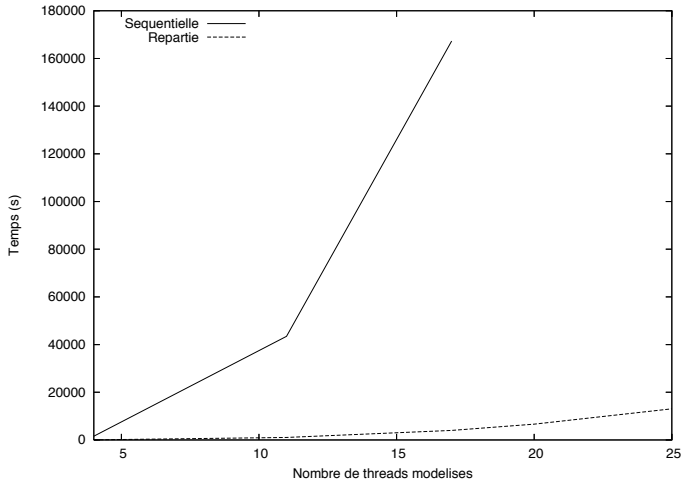


FIGURE 3.21 – Temps d'exécution en génération répartie et séquentielle pour le modèle PolyORB

moyenne 90% des états sont systématiquement envoyés sur le réseau.

3.8 Conclusion

Nous avons présenté une architecture de *model checking* réparti et parallèle, implémentée dans la LIBDMC. Cette bibliothèque offre des interfaces correspondant à la manipulation d'automates afin de permettre l'intégration d'une logique

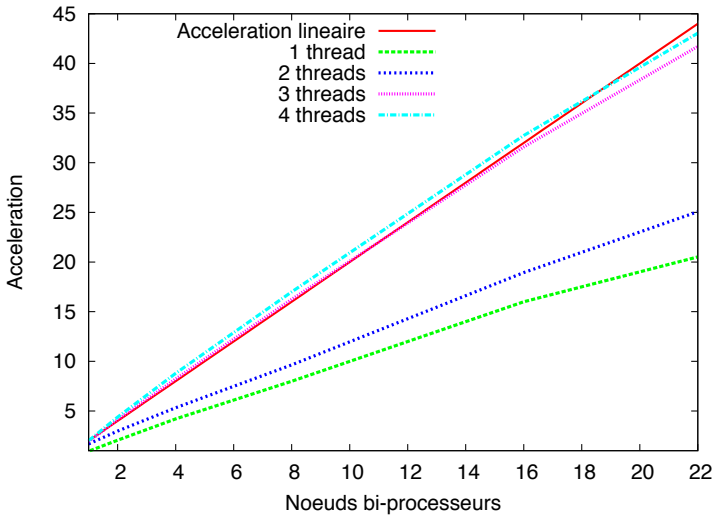


FIGURE 3.22 – Accélérations selon le nombre de *threads* pour le modèle Philosophes(14)

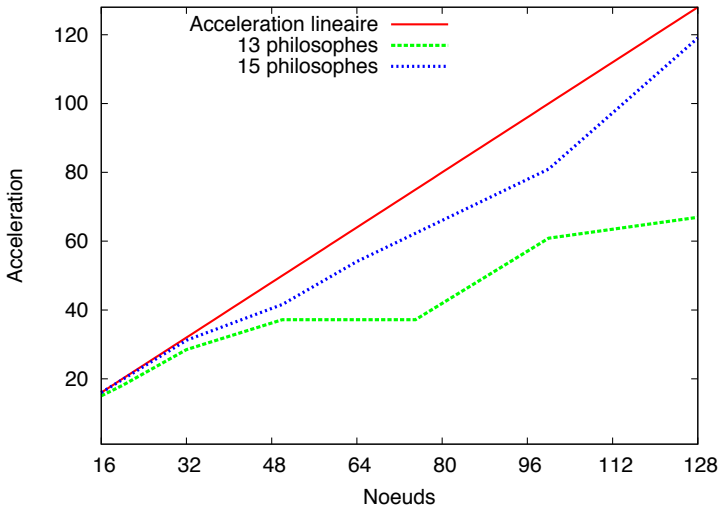


FIGURE 3.23 – Accélérations pour la génération de l'espace d'états du modèle du diner des philosophes sur le Grid'5000

de calcul d'états fournie par un *model checker* au sein d'une logique de répartition. Nous proposons un cadre de travail permettant aux concepteurs de *model checkers* de se concentrer sur leur véritable centre de préoccupation qu'est la vérification.

L'implémentation que nous avons réalisée ne dépend que de mécanismes standardisés tels que les threads POSIX ou les *sockets* TCP, et de bibliothèques très courantes, ce qui la rend portable.

Les *model checker* à répartir doivent respecter certains critères : les états doivent pouvoir être calculés indépendamment les uns des autres ; l'espace d'états doit être représentable sous forme de graphe (les états ne partageant pas d'information). De plus, ils doivent respecter des critères techniques sur la représentation en mémoire des états et doivent être *thread-safe*.

Ces conditions réunies, un utilisateur de cette architecture n'aura pas à se soucier des problèmes de répartition. Tout au plus devra-t-il se préoccuper de détails techniques comme la représentation mémoire contigüe des états ou encore d'avoir une implémentation *thread-safe* de son *model checker*. Mais nous avons bon espoir que de telles contraintes techniques ne sont pas rédhibitoires.

Les bons résultats obtenus lors de nos expériences valident de nos choix. Par exemple, le hachage statique simple que nous avons choisi permet une répartition homogène des états, impliquant une charge de travail généralement bien répartie. Les accélérations obtenues sont toujours linéaires, sauf lorsque les espaces d'états des modèles analysés deviennent trop simples à calculer la une puissance de calcul choisie.

Les raisons de ce succès sont à notre sens dues à deux facteurs :

– l'utilisation de plusieurs *threads* de génération ;

– un temps de calcul grand devant le temps de communication.

La combinaison de ces deux facteurs permet d'obtenir un recouvrement optimal des temps de communications par les temps de calcul des états. Le choix du *model checker* à paralléliser n'est donc pas innocent : il faut privilégier ceux dont les procédures de calcul sont assez coûteuses pour recouvrir les temps de communication.

Il est possible de relâcher cette contrainte : s'il on ne peut pas alourdir les temps de calcul, car ils sont de la responsabilité du *model checker*, on peut ajouter des mécanismes visant à réduire le nombre de communications, ce paramètre étant le seul sur lequel la LIBDMC peut jouer. Il s'agit à notre sens d'optimisations plus techniques et peu difficiles à mettre en place. On peut ainsi envisager un mécanisme de cache pour éviter d'envoyer des états déjà transmis ou bien un mécanisme d'agrégation des états pour maximiser l'usage du réseau. Ou encore, si les ressources de calcul sont sous-utilisées, on peut les utiliser pour compresser les états avant leur envoi sur le réseau et ainsi réduire le besoin en bande-passante.

Finalement, il s'avère que le *model checking* réparti et parallèle est viable dès lors que l'on respecte les conditions énoncées ci-dessus. Les travaux présentés dans cette partie sont validés par les expérimentations que nous avons menées et nous estimons que cette technique d'optimisation du *model checking* a toute sa

place auprès des autres. De plus elle peut-être indépendante des autres, comme nous l'avons vu ici avec les symétries de GreatSPN.

Deuxième partie

Saturation automatique

CHAPITRE 4

État de l'art et problématiques

Nous présentons dans ce chapitre l'état de l'art relatif aux diagrammes de décision et à leur manipulation dans l'optique de générer symboliquement un espace d'états.

4.1 Diagrammes de décision

Les diagrammes de décision sont une famille de structures de données dédiées à la représentation compactes de données. Leur principe est de stocker de manière partagée, sous forme de séquences, les parties communes des données que l'on veut représenter. Il en existe une multitude de variantes, chacune étant adaptée à des besoins spécifiques [Lin09].

Ils présentent plusieurs avantages. En particulier, leur taille ne dépend pas directement de la quantité de données stockées. Cette taille peut être inférieure de façon exponentielle par rapport à la quantité de données. De plus, leur principe même fait que la comparaison de deux ensembles de données se fait toujours en temps constant. Aussi, ils sont très généralement associés à un mécanisme de cache, ce qui fait qu'une évaluation sur un diagramme de décision ne sera toujours faite qu'une fois.

Nous présentons ici les diagrammes de décision les plus répandus, les BDD, ainsi que les MDD car la contribution de cette partie, la saturation automatique, s'appuie sur un mécanisme mis au point sur ces derniers. Nous évoquerons aussi les DDD, qui sont les ancêtres des SDD, que nous définissons ensuite.

Binary Decision Diagrams (BDD). Les BDD sont les premiers représentants de cette famille de structure de données. Ils sont apparus dans [Bry86]. Leur but était initialement d'encoder des fonctions booléennes sous forme d'un graphe. Très vite de nombreuses applications ont été développées autour de cette structure de données, dont [Bry92] présente quelques exemples.

Les premières applications au *model checking* se font dans [BCM+90, CMB90]. On retrouve les BDD appliqués aux réseaux de Petri dans [PRCB94]. L'utilisation des BDD permet d'atteindre des nombres d'états de l'ordre de 10^{20}, alors que les techniques plus traditionnelles plafonnent encore au million d'états. Il s'agit à l'époque d'une véritable révolution dans le domaine de la vérification.

Un BDD [1] est une représentation canonique d'une formule booléenne, représentée sous forme d'un graphe acyclique avec un ordre total sur les variables de la formule. Le domaine des variables est $\mathbb{B}$. Par exemple, la formule $(a \wedge b) \vee (c \wedge d)$, en utilisant l'ordre $a < b < c < d$, est représentée par le BDD de la figure 4.1. Selon une suite d'affectations aux variables, on peut déterminer si ces affectations satisfont la formule en parcourant le BDD de sa racine jusqu'aux feuilles. Par exemple, la suite d'affectations $a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1$ amène à une feuille (un « terminal ») portant la valeur 1, ce qui signifie que cette séquence est reconnue par la formule. Les séquences menant à 0 ne sont pas reconnues (en réalité, elles ne sont pas représentées afin d'économiser en espace mémoire). Enfin, un nœud menant au même successeur quelque soit les valeurs est supprimé, ses pères étant reliés à son successeur.

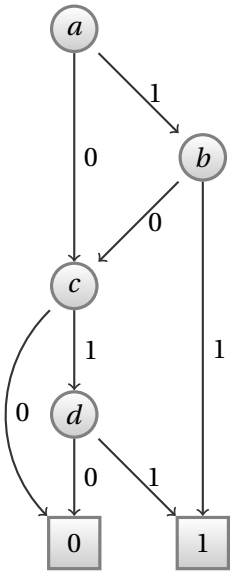


FIGURE 4.1 – BDD de la formule $(a \wedge b) \vee (c \wedge d)$

Leur application au *model checking* est quasiment immédiate [BCM+90] : les variables sont celles qui encodent le système. Si jamais les valeurs que peuvent atteindre les variables sont dans $\mathbb{N}$, alors il suffit d'utiliser plusieurs variables pour encoder une valeur non booléenne.

1. En réalité, il s'agit des *Reduced Ordered BDD (ROBDD)*, mais *BDD* est le terme consacré dans la littérature

Ajouter un état au système encodé par un BDD à k variables passe par l'application d'un BDD à $2k$ variables sur le BDD représentant l'espace d'états. Ce BDD à $2k$ variables représente les relations de transitions s'appliquant au système. Chaque relation de transition indique pour une variable du système sa valeur avant application de la transition et celle après application, d'où les $2k$ variables.

Multi-valued Decision Diagrams (MDD). Les MDD sont initialement introduits par [SHMB90] et seront repris par [MC99] pour stocker les espaces d'états de réseaux de Petri. Par rapport aux BDD, ces diagrammes de décision étendent le domaine des variables à $\mathbb{N}$, mais ce domaine doit être borné.

Nous présentons plus en détail leur manipulation en section 4.4 pour expliquer la saturation.

Data Decision Diagrams (DDD). Les DDD [CEPA+02] sont très souples : les variables ont aussi leurs domaines dans $\mathbb{N}$, mais ces domaines sont non-bornés. Les séquences (ou « chemins ») peuvent être de tailles variables et aucun ordre n'est imposé sur les variables. Il est aussi possible de répéter une même variable sur un chemin. À la différence des BDD ou MDD, les relations de franchissement sont encodées non pas sous forme de diagrammes de décision, mais à l'aide d'opérations appelées « homomorphismes ».

Hierarchical Set Decision Diagrams (SDD). Les travaux présentés dans cette partie étant basés sur les SDD, nous les définissons intégralement dans la section suivante.

4.2 Hierarchical Set Decision Diagrams : SDD

Pour étendre les DDD, [TM04, CTM05] introduisent les *Hierarchical Set Decision Diagrams* (SDD), les diagrammes de décision hiérarchiques. Ils conservent les propriétés des DDD telles que les chemins de longueur variable ou encore l'absence de bornes sur les valeurs portées sur les arcs. Ils utilisent aussi comme mécanisme de manipulation les homomorphismes.

Nous présentons dans un premier temps les SDD eux-mêmes (section 4.2.1), puis les homomorphismes (section 4.2.2).

4.2.1 La structure de données

Les SDD sont des diagrammes de décision dans lesquels les arcs sont étiquetés par un ensemble de valeurs au lieu d'une seule valeur. Cet ensemble peut-être lui-même un SDD ce qui rend les SDD hiérarchiques.

Ils représentent des ensembles de séquences d'affectation d'ensemble de la forme $\omega_1 \in s_1; \omega_2 \in s_2; \ldots; \omega_n \in s_n$ où les ω_i sont des variables et les s_i des ensembles de valeurs.

Aucun ordre sur les variables n'est imposé, et la même variable peut apparaître plusieurs fois dans une séquence d'affectations. Le terminal $\boxed{1}$ représente la séquence d'affectation vide qui termine toutes les séquences valides. Le terminal $\boxed{0}$ termine les séquences non valides et représente l'ensemble vide des séquences d'affectation. Par la suite, Var représente un ensemble de variables et, quel que soit $\omega \in Var$, $Dom(\omega)$ représente le domaine de ω qui peut-être infini.

Les SDD sont définis comme suit :

Définition 4.1 : Diagrammes de décision hiérarchiques
L'ensemble $\mathbb{S}$ des SDD est défini inductivement par $\delta \in \mathbb{S}$ si :
- $\delta \in \{\boxed{0}, \boxed{1}\}$; ou
- $\delta = \langle \omega, \pi, \alpha \rangle$ tel que :
  - $\omega \in Var$
  - $\pi = \{s_0, \ldots, s_n\}$ est une partition finie de $Dom(\omega)$
  - $\alpha : \pi \mapsto \mathbb{S}$, tel que $\forall i \neq j, \alpha(s_i) \neq \alpha(s_j)$

  De plus, $\forall \omega \in Var, \forall s \in 2^{Dom(\omega)}, \omega \xrightarrow{s} \boxed{0} \equiv \boxed{0}$

Par convention, quand il existe, l'élément de la partition π qui mène au SDD $\boxed{0}$ n'est pas représenté. Nous notons $\omega \xrightarrow{s} \delta'$ le SDD $\delta = \langle \omega, \pi, \alpha \rangle$ avec $\pi = \{s\}$ et $\alpha(s) = \delta'$.

Cette définition a plusieurs conséquences :
- les SDD supportent les domaines de taille infinie. Par exemple, si $Dom(\omega) = \mathbb{R}$, on peut utiliser la partition $\{]0 \ldots 3],]3, \infty]\}$;
- des SDD ou autres types d'ensembles peuvent être utilisés pour les domaines de variable, introduisant ainsi de la hiérarchie ;
- les SDD peuvent gérer des chemins de longueur variable. Cette caractéristique est utile lorsque l'on veut représenter des structures dynamiques telles que des files ou des listes.

Cette définition assure qu'il existe une représentation unique et canonique pour chaque ensemble de séquences d'affectation. La taille finie de la partition π assure qu'α peut être stockée sous forme d'un ensemble de paires $\langle s_i, \delta_i \rangle$ et donc que π peut aussi être définie implicitement par α.

Pour gérer les chemins de longueur variable, les SDD doivent représenter des ensembles de séquences « compatibles ». Une opération sur les SDD est dite partiellement définie si elle peut produire des séquences incompatibles :

Définition 4.2 : Séquences compatibles
Une séquence de SDD est un SDD de la forme $\omega_0 \xrightarrow{s_0} \ldots \omega_n \xrightarrow{s_n} \boxed{1}$. Soit σ_0 et σ_1 deux séquences, σ_0 et σ_1 sont compatibles (noté $\sigma_0 \approx \sigma_1$) si et seulement si :
– $\sigma_0 = \boxed{1} \wedge \sigma_1 = \boxed{1}$
– $\sigma_0 = \omega \xrightarrow{s} \delta \wedge \sigma_1 = \omega' \xrightarrow{s'} \delta'$ tel que $\begin{cases} \omega = \omega' \\ s \approx s' \\ s \bigcap s' \neq \emptyset \Rightarrow \delta \approx \delta' \end{cases}$

La définition de la compatibilité des ensembles portés sur les arcs dépend entièrement de ces ensembles. Elle est donc laissée à l'appréciation de l'utilisateur. Pour les SDD, il suffira évidemment de reprendre la définition ci-dessus. Deux BDD seront compatibles s'ils ont la même hauteur et le même ordre de variables.

Si cette notion de séquence compatible semble restrictive, elle l'est en fait moins que ce que l'on trouve habituellement pour les diagrammes de décision, où la norme est d'utiliser un ensemble fixe de variables, dans un ordre fixé pour tous les chemins.

Dans la pratique, cette définition de séquence compatible sert à encoder des structures dynamiques telles que les files. Pour encoder une file, la même variable ω est répétée. La dernière occurrence de ω sur tous les chemins sera étiquetée par un marqueur spécial, noté $\sharp$. Ainsi, $\omega \xrightarrow{\sharp} \boxed{1}$ représente une file vide et $\omega \xrightarrow{s_1} \omega \xrightarrow{\sharp} \boxed{1}$ est une file avec un seul élément, où ω prend ses valeurs dans s_1.

Les SDD supportent les opérations ensemblistes habituelles (union $\bigcup$, intersection $\bigcap$, différence $\setminus$). Ils offrent aussi la concaténation $\delta_1.\delta_2$ qui remplace le terminal $\boxed{1}$ de δ_1 par δ_2, ce qui correspond à un produit cartésien.

Les SDD sont créés soit à l'aide de la concaténation, soit à l'aide de l'union. Dans le premier cas, il n'est pas nécessaire de procéder à une canonisation car les opérandes sont déjà canonisés et le seul nœud touché est le terminal $\boxed{1}$ de la première opérande de la concaténation. Le résultat reste donc canonique.

Les SDD sont donc canonisés à la construction à l'aide de l'union. La canonicité des SDD est due à deux propriétés : π est une partition et deux arcs provenant d'une même variable ne peuvent pointer sur le même SDD. Donc n'importe quelle valeur $x \in Dom(\omega)$ est représentée sur au plus un arc ; ainsi la construction de $\omega \xrightarrow{s} \delta \cup \omega \xrightarrow{s'} \delta$ donne $\omega \xrightarrow{s \cup s'} \delta$. Les arcs sont donc fusionnés dans ce cas (figure 4.2).

Un effet opposé est obtenu à la construction d'un SDD à partir de deux SDD dont deux arcs $\langle s, \delta \rangle$ et $\langle s', \delta' \rangle$ sont en intersection : $s \bigcap s' \neq \emptyset$. Il faut alors créer trois arcs $\langle s \setminus s', \delta \rangle$, $\langle s' \setminus s, \delta' \rangle$ et $\langle s \bigcap s', \delta \bigcup \delta' \rangle$ (figure 4.3).

Cette procédure de canonisation est coûteuse. En effet, elle nécessite de s'assurer que les ensembles portés sur les arcs d'un nœud de variable ω forment bien une partition de $Dom(\omega)$. Donc pour faire l'union de deux SDD, il faut comparer

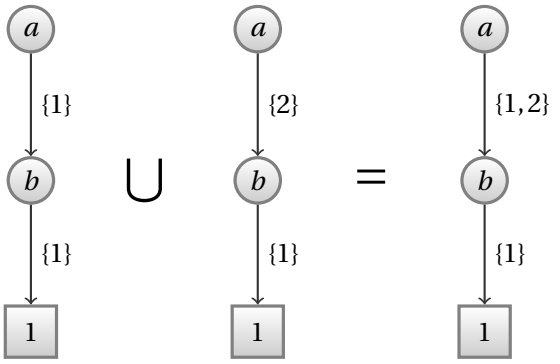


FIGURE 4.2 – Fusion des arcs

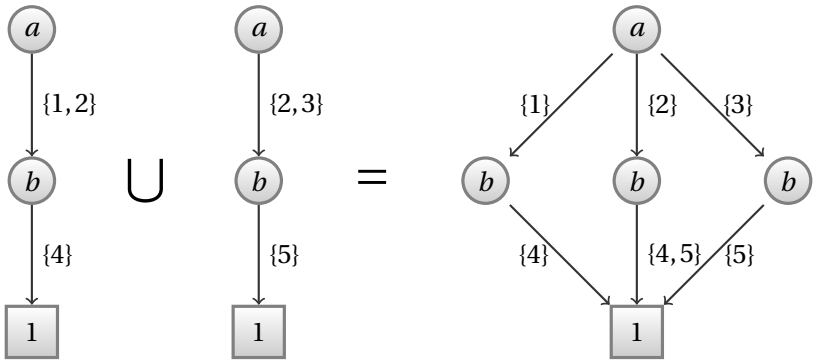


FIGURE 4.3 – Découpage des arcs

chaque arc de l'un avec chaque arc de l'autre, afin de déterminer la partition π du SDD résultant. La complexité de la canonisation est donc quadratique au niveau d'une variable. De plus, l'union doit se propager inductivement sur les SDD opérandes, répercutant ainsi cette complexité sur chaque nœud rencontré.

Les travaux présentés dans le chapitre 5 ont entre autres pour but de mettre en place des mécanismes minimisant le coût de cette canonisation en évitant de créer des SDD temporaires.

4.2.2 Manipulation des SDD

Opérations ensemblistes mises à part, les SDD se manipulent avec des opérations spécifiques, les **homomorphismes**. Deux types d'homomorphismes peuvent être appliqués : les basiques et les inductifs.

Un homomorphisme basique est une application $\Phi : \mathbb{S} \mapsto \mathbb{S}$ satisfaisant $\Phi(\boxed{0}) = \boxed{0}$ et $\forall \delta, \delta' \in \mathbb{S}, \Phi(\delta + \delta') = \Phi(\delta) + \Phi(\delta')$. La somme et la composition de deux homomorphismes sont des homomorphismes.

Quelques homomorphismes standards sont définis : l'homomorphisme identité Id, l'intersection $(*)$, l'union $(+)$, la concaténation $(.)$ et le point fixe$(^\star)$. Par

exemple, l'homomorphisme $\delta * Id$, où $\delta \in \mathbb{S}$, permet de sélectionner les séquences appartenant à δ. Les homomorphismes $\delta.Id$ et $Id.\delta$ permettent d'effectuer une concaténation à gauche ou à droite de séquences. La concaténation à gauche est souvent utilisée pour ajouter une affectation à une séquence, elle est notée $\omega \xrightarrow{s} Id$.

Pour chaque homomorphisme h et chaque $\delta \in \mathbb{S}$, $h^\star(\delta)$ est évalué en répétant $\delta \leftarrow h(\delta)$ jusqu'à ce qu'un point fixe soit atteint. Autrement dit, $h^\star(\delta) = h^n(\delta)$ où $n \in \mathbb{N}$ est le plus petit entier tel que $h^n(\delta) = h^{n+1}(\delta)$. Cet opérateur est très souvent appliqué à $(h + Id)$, permettant ainsi d'accumuler les séquences d'affectation nouvellement calculées.

Les **homomorphismes inductifs** permettent aux utilisateurs de SDD d'écrire leurs propres opérations sur $\mathbb{S}$. Un homomorphisme inductif ϕ est défini par son évaluation sur le terminal $\boxed{1}$ et son évaluation $\phi(\omega, s)$ pour chaque $\omega \in Var$ et chaque $s \subseteq Dom(\omega)$. L'expression $\phi(\omega, s)$ est elle-même un homomorphisme (possiblement inductif) qui sera appliqué sur le nœud successeur $\alpha(s)$. Le résultat de $\phi(\langle \omega, \pi, \alpha \rangle)$ est alors défini par $\sum_{s \in \pi} \phi(\omega, s)(\alpha(s))$ où $\sum$ représente l'union.

Les homomorphismes sont des opérations linéaires. Il faut donc que les manipulations des ensembles portés sur les arcs respectent cette linéarité. Cela signifie qu'elles doivent être indépendantes des partitions. Par exemple, il est impossible de rechercher un maximum parmi toutes les valeurs contenues dans les ensembles.

Pour illustrer les homomorphismes inductifs, considérons l'homomorphisme 4.1 Inc, paramétré par une variable v et une valeur n. Son but est d'incrémenter toutes les valeurs inférieures à n des ensembles portés sur les arcs partant de la variable v. Son application sur $\omega \xrightarrow{s}$ peut produire deux résultats. Soit la variable v est rencontrée auquel cas les valeurs sur l'arc sont modifiées si elles satisfont à la condition d'être inférieures à n. Le résultat de cette modification est concaténé en tête de la suite de la séquence à l'aide de Id. Si la variable v n'est pas rencontrée, alors on propage l'application de Inc sur le successeur. Pour ce faire, on retourne simplement l'homomorphisme inductif à appliquer sur la suite de la séquence.

Homomorphisme 4.1 : Inc

$$
\begin{aligned}
Inc(v,n)(\omega \xrightarrow{s}) &= \begin{cases} \omega = v : \omega \xrightarrow{\{x+1 | x \in s \wedge x < n\}} Id \\ \omega \neq v : \omega \xrightarrow{s} Inc(v,n) \end{cases} \\
Inc(v,n)(\boxed{1}) &= \boxed{1}
\end{aligned}
$$

Ainsi, l'application de $Inc(b,4)$ sur le SDD δ_0 de la figure 4.4 donnera le SDD δ_1 de la figure 4.5 : les valeurs de la variable a ne sont pas modifiées et seules les valeurs de b strictement inférieures à 4 sont incrémentées.

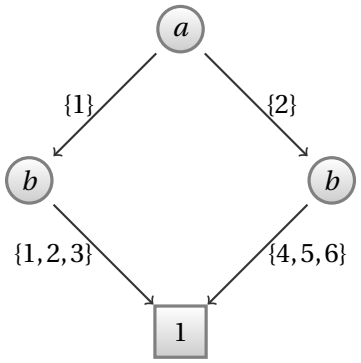


FIGURE 4.4 – SDD δ_0

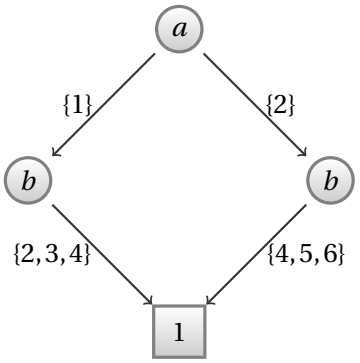


FIGURE 4.5 – SDD $\delta_1 = Inc(b,4)(\delta_0)$

Pour appliquer un homomorphisme sur un SDD imbriqué dans un niveau de hiérarchie, il faut passer par un homomorphisme intérmediaire. Par exemple l'homomorphisme 4.2 se propage jusqu'à la variable v. Lorsqu'il la rencontre, alors il applique l'homomorphisme $Inc(w,n)$ sur le SDD imbriqué sur l'arc. Ainsi, l'application de $Inc_{Hier}(x,b,4)$ sur le SDD δ_2 de la figure 4.6 donne le SDD δ_3 de la figure 4.7.

Homomorphisme 4.2 : Incrémentation hiérarchique

$$Inc_{Hier}(v,w,n)(\omega \xrightarrow{s}) = \begin{cases} \omega = v : & \omega \xrightarrow{Inc(w,n)(s)} Id \\ \omega \neq v : & \omega \xrightarrow{s} Id \end{cases}$$

$$Inc_{Hier}(v,w,n)(\boxed{1}) = \boxed{1}$$

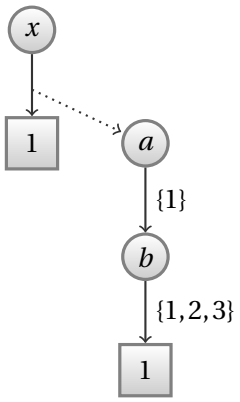


FIGURE 4.6 – SDD δ_2

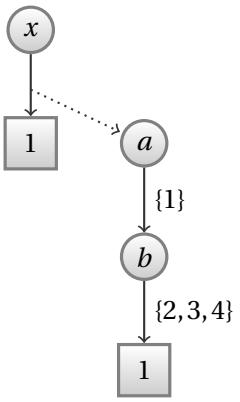


FIGURE 4.7 – SDD $\delta_3 = Inc_{Hier}(x,b,4)$

Il est à noter que toutes les évaluations d'homomorphismes sont conservées dans un cache. Celui-ci est en effet impératif afin d'éviter de refaire des calculs, ce qui est une situation fréquente lors de l'usage des homomorphismes. Ce détail est important car conserver dans ce cache des applications n'ayant lieu qu'une fois (donc créant des nœuds temporaires) peut amener à une utilisation de la mémoire trop grande. Nous verrons par la suite que cela peut-être évité grâce à la saturation.

Notation
L'ensemble des homomorphismes sera noté $\boldsymbol{H}$.

4.3 Génération d'un espace d'état

Nous comparons dans cette section diverses manières de générer un espace d'états en utilisant les diagrammes de décision (4.3.1). Ensuite nous montrons comment générer l'espace d'états d'un réseau de Petri en utilisant les SDD et les homomorphismes (4.3.2).

4.3.1 Comparaisons de relations de franchissement

Si calculer un espace d'états symboliquement est un simple point fixe sur un ensemble d'événements à appliquer, il existe plusieurs manières d'évaluer cette boucle. Nous en montrons ici trois différentes [2].

Algorithme 4.1 : Accessibilité standard

Données :
T : l'ensemble des événements
s_0 : l'ensemble des états initiaux
$nouveaux$: les nouveaux états restant à explorer
S : les états déjà atteints

```
1 nouveaux ← s_0
2 S ← s_0
3 tant que nouveaux ≠ ∅ faire
4     tmp ← T(nouveaux)
5     nouveaux ← tmp \ S
6     S ← S ∪ tmp
```

La manière « naïve » (algorithme 4.1) correspond à la méthode habituellement utilisée pour la génération d'un espace d'états explicite [3] : un ensemble d'états $nouveaux$ ne contient que les états encore non-explorés. L'inconvénient majeur

2. Nous utilisons un léger abus de notation en utilisant $T(x)$ pour indiquer $(\sum_{t\in T} t)(x)$)
3. Sans diagrammes de décision, que nous avons vue dans la première partie (algorithme 2.1)

de cette méthode est le fait d'utiliser la différence, engendrant plus de calculs qui sont d'un intérêt limité pour le résultat final.

Algorithme 4.2 : Boucle symbolique standard

Données :
T : l'ensemble des événements
s_0 : l'ensemble des états initiaux
$nouveaux$: les nouveaux états restant à explorer
S : les états déjà atteints

```
1 nouveaux ← s_0
2 S ← ∅
3 tant que nouveaux ≠ S faire
4     nouveaux ← (T + Id)(nouveaux)
```

L'algorithme 4.2 applique directement l'ensemble de la relation de franchissement à l'ensemble des états disponibles, ce qui est en pratique bien plus efficace que l'algorithme 4.1. Cela est dû au fait que la taille d'un diagramme de décision n'est pas directement liée au nombre d'états encodés, et donc que la taille de $nouveaux$ dans l'algorithme 4.1 peut être bien plus grande. De plus, en utilisant un cache, le fait d'appliquer à nouveau la relation de transition sur des états déjà explorés ne sera pas coûteux.

Cet algorithme est similaire au point fixe que l'on trouve dans [BCM+90]. La manière habituelle est d'encoder la relation de transition en utilisant des diagrammes de décision avec deux variables (une représentant l'état avant la transition et l'autre après) pour chaque variable du diagramme de décision d'un état. Garder chacune de ces transitions isolées l'une de l'autre implique un surcoût en temps car plusieurs transitions ne peuvent pas partager le parcours du diagramme de décision. L'union T des relations de transitions est donc encodée sous la forme d'un seul diagramme de décision. Cependant, le simple fait de calculer T peut s'avérer impossible dans certains cas, amenant à des algorithmes plus élaborés tels que ceux trouvés dans [BCL91]. Ce dernier encode la relation de transition par partie à l'aide de plusieurs BDD, et non plus d'un seul.

On retrouve une approche similaire qui est de chaîner l'application d'agrégats de transitions [RCP95] (algorithme 4.3). Ces agrégats sont créés en essayant de minimiser les dépendances entre eux. Ils sont encodés à l'aide de diagrammes de décision plus petits que si l'on avait encodé toutes les transitions ensembles. Ces agrégats sont ensuite appliqués les uns à la suite des autres, ce qui permet d'explorer plus rapidement l'espace d'états. Bien que cette variante repose sur une heuristique, elle a été empiriquement montrée comme meilleure que l'algorithme 4.2.

Finalement, la saturation [CMS03] est empiriquement meilleure, en ordres de grandeurs, que le chaînage. Nous reviendrons en détail sur ce mécanisme en section 4.4, puisqu'il est le fondement des travaux présentés au chapitre 5.

Algorithme 4.3 : Boucle de chaînage de transitions

Données :
A : l'ensemble des agrégats d'événements
s_0 : l'ensemble des états initiaux
$nouveaux$: les nouveaux états restant à explorer
S : les états déjà atteints

```
1 nouveaux ← s_0
2 S ← ∅
3 tant que nouveaux ≠ S faire
4     S ← nouveaux
5     pour chaque a ∈ A faire
6         nouveaux ← (a + Id)(nouveaux)
```

4.3.2 Réseaux de Petri place/transition

Dans cette section, nous montrons comment sont utilisés les SDD pour générer l'espace d'états d'un modèle. Nous utiliserons à cette fin les réseaux de Petri place/transition, définis comme suit :

Définition 4.3 : Réseau de Petri places/transitions
Un réseau de Petri place/transition R est un n-uplet $\langle P, T, Pre, Post, M_0 \rangle$ où :
– P est un ensemble fini de places ;
– T est un ensemble fini de transitions ;
– Pre, la matrice d'incidence arrière définie par $Pre : P \times T \to \mathbb{N}$;
– $Post$, la matrice d'incidence avant définie par $Post : P \times T \to \mathbb{N}$;
– $M_0 \subset \mathbb{N}^P$ est le marquage initial du réseau.
$Pre(p,t)$ contient la valeur entière n associée à l'arc allant de p à t et $Post(p,t)$ contient la valeur entière n associée à l'arc allant de t à p
Une transition $t \in T$ est dite « franchissable » à partir d'un marquage M si et seulement si $\forall p \in P, M(p) \geq Pre(p,t)$. Si t est franchissable, son franchissement mène au marquage M' :

$$\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t)$$

On note ${}^\bullet t$ l'ensemble des places p_i tel que $Pre(p_i,t) > 0$ et $t^\bullet$ l'ensemble des places p_j tel que $Post(p_j,t) > 0$.

L'ensemble des états accessibles d'un réseau de Petri est calculé en effectuant la fermeture transitive du marquage initial par la relation de franchissement décrivant ce réseau. Nous montrons ici comment est écrit cette relation.

La figure 4.8 présente la modélisation en réseaux de Petri d'un système simple de client-serveur. Le client C envoie une requête asynchrone au serveur S, puis attend une réponse de ce même serveur après traitement du message par ce dernier (place S_2), toujours de manière asynchrone.

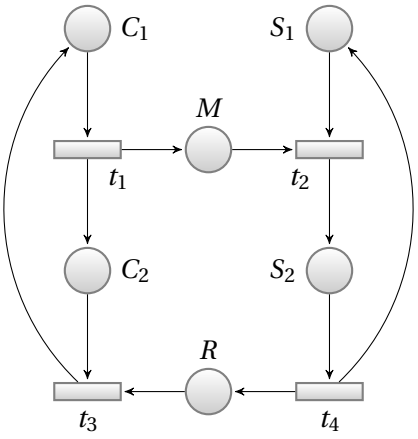
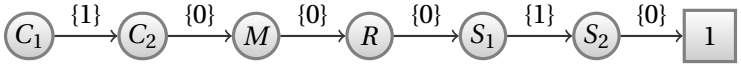


FIGURE 4.8 – Modèle d'un client-serveur en réseaux de Petri

La première étape consiste à en définir l'espace d'états initial, donc le marquage initial de chaque place. Plusieurs encodages sont possibles : on peut considérer par exemple que les variables représentent les jetons, et la valeur de ces jetons indique leurs positions dans le réseau de Petri. Nous utiliserons ici les places comme variables, les valeurs de ces dernières seront donc les marquages des places. Si nous positionnons le marquage de C_1 et de S_1 à 1 et de toutes les autres places à 0, un SDD M_0 correspondant au marquage initial est :



Il existe autant de SDD possibles pour cet encodage qu'il y a d'ordres de variables possibles. Une fois ce marquage défini, il faut mettre en place la relation de franchissement, qui se base sur les transitions du réseau de Petri. À cette fin, nous utilisons deux homomorphismes h^- et h^+ :

Homomorphisme 4.3 : h^-

$$h^-(p,m)(\omega \xrightarrow{s}) = \begin{cases} \omega = p : \omega \xrightarrow{\{n-m \mid n \in s \wedge n \geq m\}} Id \\ \omega \neq p : \omega \xrightarrow{s} h^-(p,m) \end{cases}$$

$$h^-(p,m)(\boxed{1}) = \boxed{0}$$

Homomorphisme 4.4 : h^+

$$h^+(p,m)(\omega \xrightarrow{s}) = \begin{cases} \omega = p:\omega \xrightarrow{\{n+m|n\in s\}} Id \\ \omega \neq p:\omega \xrightarrow{s} h^+(p,m) \end{cases}$$
$$h^+(p,m)(\boxed{1}) = \boxed{0}$$

La relation de franchissement est décrite arc par arc du réseau de Petri. À chaque transition t est associée une série d'homomorphismes h^+ et h^-.

L'homomorphisme $h^-(p,m)$ a pour rôle de sélectionner les marquages $M(p)$ de la place p tel que $M(p) \geq Pre(p,t)$ et de mettre à jour le marquage de p par $M(p) - Pre(p,t)$ si le franchissement est autorisé. Si jamais $\{n-m|n \in s \wedge n \geq m\}$ renvoie un ensemble vide, alors cela signifie que la séquence n'est pas reconnue. Dans ce cas, cela implique qu'elle se termine par $\boxed{0}$, et donc qu'elle n'est pas représentée. L'homomorphisme $h^+(p,m)$ augmente le marquage de la place p de m jetons.

Pour décrire complètement une transition d'un réseau de Petri, il faut composer l'ensemble des homomorphismes h^+ et h^- la décrivant. La transition t est encodée par l'homomorphisme h_t suivant ($\bigcirc_{x\in X}$ représentant les compositions successives de tous les x) :

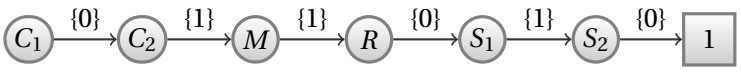
$$h_t = \bigcirc_{p_i \in t^\bullet} h^+(p_i, Post(p_i,t)) \circ \bigcirc_{p_i \in {}^\bullet t} h^-(p_i, Pre(p_i,t))$$

Les homomorphismes h^- sont appliqués en premier afin de sélectionner tous les marquages permettant de franchir t et de retirer les jetons dans les places en entrée. Ensuite les homomorphismes h^+ augmentent le marquage des places en sortie.

Par exemple, l'homomorphisme décrivant la transition t_1 de la figure 4.8 est le suivant :

$$h_{t_1} = h^+(C_2,1) \circ h^+(M,1) \circ h^-(C_1,1)$$

L'application de cet homomorphisme sur M_0 donne :



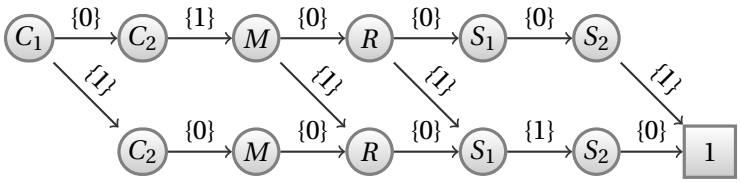
Il faut ensuite appliquer l'ensemble des transitions. Pour cela, il suffit de faire l'union des homomorphismes décrivant les transitions et de les appliquer dans une fermeture transitive :

$$H = (\sum_{t_i \in T} h_{t_i} + Id)^\star$$

L'homomorphisme complet pour le modèle de la figure 4.8 est donc le suivant :

$$H_{cs} = (h_{t_1} + h_{t_2} + h_{t_3} + h_{t_4} + Id)^\star$$

L'application de H_{cs} sur M_0 donne le SDD suivant qui encode les 4 états possibles du système :



Nous avons présenté ici des homomorphismes permettant de traiter des réseaux de Petri place/transition. Il est possible de traiter des cas plus complexes, tels que des arcs inhibiteurs ou des capacités. Ces extensions sont décrites en détail dans [CEPA+02].

La forme de cette relation de franchissement finale n'est pas spécifique aux réseaux de Petri. Elle correspond à ce que l'on trouve dans l'algorithme 4.2 : une fermeture transitive d'événements. Nous verrons au chapitre suivant comment nous obtenons automatiquement un effet de saturation en nous appuyant sur cette forme générale.

4.4 Saturation

Un problème propre au *model checking* symbolique est l'effet de « pic ». Le calcul du diagramme de décision représentant l'espace d'états final d'un système passe par la génération d'un grand nombre d'états dont les encodages en diagrammes de décision ne seront pas présents dans la représentation finale. Cela peut empêcher la terminaison du calcul en utilisant trop de mémoire.

Il existe plusieurs manières de s'attaquer à ce problème. La première possibilité est de considérer les variables : leur ordre influe sur cet effet de pic, ainsi que sur la taille de la représentation finale. Cependant, ce problème est connu pour être NP-complet [BW96]

Il est aussi possible de travailler sur l'encodage du système (par exemple, l'encodage d'un réseau de Petri place/transition peut se faire en utilisant les places ou les jetons comme variable). Citons par exemple [CLY07] qui utilise les invariants d'un réseau de Petri afin de fusionner des variables.

Une autre solution est de chercher à appliquer de la manière la plus efficace les événements qui s'appliquent sur le système vérifié. Nous avons ainsi vu en section 4.3 différentes manières d'appliquer les événements du système vérifié. Citons aussi [GV01] qui bloque l'évolution d'une variable jusqu'à obtenir un point fixe sur le reste du diagramme de décision, répétant le processus sur toutes les variables jusqu'à la génération complète.

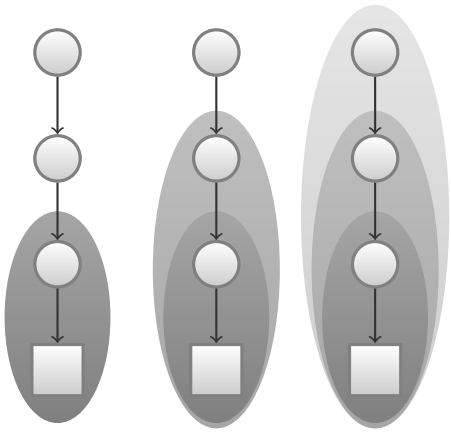


FIGURE 4.9 – Illustration schématique de la saturation

La saturation fait partie de cette classe de solutions. L'idée de base est de calculer des points fixes locaux (« saturer »), en partant des feuilles d'un diagramme de décision pour remonter jusqu'à la racine, tout en re-saturant les nœuds inférieurs dès qu'un nœud est modifié (figure 4.9).

Nous exposons dans cette section le mécanisme de saturation. Nous le présentons d'abord pour les MDD, puis pour les SDD.

4.4.1 MDD

La saturation est apparue initialement dans [CLS01b]. La première version nécessitait de créer une partition du modèle d'entrée, un réseau de Petri, de manière à ce que chaque partie possède un espace d'états local dont on puisse déterminer les états potentiels, calculés explicitement avant la génération globale. Il faut donc que ces états potentiels soient en nombre finis.

Nous nous basons sur [CMS03] qui s'affranchit de cette contrainte en découvrant à la volée les états locaux. Ce mécanisme est implémenté dans le *model checker* SMART [CM04].

Le principe général est de ne pas considérer les relations de transition comme étant monolithiques. Au contraire, il faut créer des partitions de ces relations. [CLS01b] s'appuie pour cela sur une représentation de Kronecker [Pla85] [4]. Ce découpage a pour but d'identifier les parties d'une relation s'appliquant aux mêmes variables du diagramme de décision et de les représenter selon une forme disjonctive : une somme d'événements disjoints pouvant être appliqués indépendamment.

Ce découpage implique que l'espace d'états peut être généré en tirant les événements du système dans n'importe quel ordre, à la condition que chaque événement soit appliqué jusqu'à obtention d'un point fixe. Cette liberté permet de sé-

4. Cet encodage est inspiré de ce qui se fait pour les modèles stochastiques

lectionner les événements à tirer exhaustivement pour qu'un nœud de MDD soit **saturé**. Cela signifie qu'il a atteint un état tel qu'il est impossible de le faire évoluer tant que d'autres événements plus globaux au système n'ont pas été tirés. Ces événements sont ceux qui touchent les autres nœuds. Ensuite, l'évaluation étant faite en profondeur, le fait qu'un nœud soit saturé indique que tous ses successeurs le sont.

Une caractéristique d'un nœud saturé est qu'il y a de bonnes chances qu'il existe dans l'espace d'états final, alors qu'un nœud non saturé ne le sera jamais. C'est pourquoi l'évaluation d'un sous-ensemble d'événements est tiré exhaustivement jusqu'à stabilisation.

Concrètement, les composantes du modèle sont agrégées en sous-systèmes, chacun avec leurs espaces d'états locaux potentiels. Chaque variable du MDD représente un des ces sous-systèmes. Les valeurs des variables indiquent l'index d'un état dans l'espace d'états local. L'espace d'états potentiels global est donc vu comme un produit cartésien des espaces d'états de ses composantes.

La relation est éclatée par événement et par niveau d'application (c'est à dire par variable dans le MDD) pour obtenir une forme disjonctive. Celle-ci est ensuite encodée par un ensemble de matrices creuses [5]. Chaque matrice représente l'évolution de l'espace d'états d'un sous-système.

Ces matrices nécessitent de connaître à l'avance les états potentiels que peuvent atteindre les sous-systèmes. En effet, une entrée à 1 dans une matrice à la ligne i et à la colonne j indique qu'il est possible de passer de l'état i à l'état j du sous-système correspondant. Dans [CLM07], les auteurs utilisent par simplification les invariants des réseau de Petri pour en déduire les bornes des marquages. Cela leur permet de connaître les états potentiels de chaque sous-système. Dans [CMS03], ces matrices sont mises à jour à la volée lors de la découverte de nouveaux états locaux d'un sous-système, ce qui est un processus complexe.

Une fois la relation de transition définie, la saturation est effectuée en suivant l'algorithme de la figure 4.10, que nous avons repris tel quel de [CLM07].

Les deux routines importantes de cet algorithme sont $Saturate$ et $Fire$, les autres étant des fonctions utilitaires pour effectuer une recherche dans une table de hachage, construire un MDD, etc. Ces deux routines sont mutuellement récursives.

$Saturate$ a pour rôle de saturer un nœud p déjà existant. Elle commence par saturer tous les successeurs en appellant $Fire$, puis sature le niveau courant en le modifiant directement. Quant à $Fire$, elle calcule les nouveaux nœuds successeurs de p, qu'elle sature en appelant récursivement $Saturate$ avant de les rendre à la routine appelante.

Les résultats rapportés dans [CLM07] sont excellents. Par exemple, la génération de l'espace d'état de la modélisation d'un protocole de verrouillage en anneau s'exécute en 0,4s alors que NuSMV prend plus de 8000s et Cadence SMV plus de 11000s. Le diner des philosophes pour 100 philosophes s'exécute en 0,4s contre

5. Ces matrices sont en réalité encodées à l'aide de MDD à $2k$ variables, à la manière des BDD

Generate(in s : array $[1..K]$ of *lcl*) : *node*

Build an MDD rooted at r, in $UT[K]$, encoding $\mathcal{N}^*_{\mathcal{E}}(\mathbf{s})$ and return r.

1. declare $r,\ p$: *node*;
2. declare l : *lvl*;
3. $p \leftarrow 1$;
4. for $l = 1$ to K do
5. $\quad r \leftarrow NewNode(l)$;
6. $\quad r[\mathbf{s}[l]] \leftarrow p$;
7. $\quad Saturate(r)$;
8. $\quad p \leftarrow r$;
9. return r;

Saturate(in p : *node*)

Saturate node p and put it in $UT[p.lvl]$.

1. declare α : *evnt*;
2. declare $f,\ u$: *node*;
3. declare $i,\ j$: *lcl*;
4. declare $\mathcal{F}$: set of *evnt*;
5. $\mathcal{F} \leftarrow \mathcal{E}_{p.lvl}$;
6. while $\mathcal{F} \neq \emptyset$ do
7. $\quad \alpha \leftarrow Pick(\mathcal{F})$;
8. $\quad$ foreach $i \in Locals(\alpha, p)$ do
9. $\quad\quad f \leftarrow Fire(\alpha, p[i])$;
10. $\quad\quad$ foreach $j \in \mathcal{N}_{p.lvl,\alpha}(i)$ do
11. $\quad\quad\quad u \leftarrow Union(f, p[j])$;
12. $\quad\quad\quad$ if $u \neq p[j]$ then
13. $\quad\quad\quad\quad p[j] \leftarrow u$;
14. $\quad\quad\quad\quad \mathcal{F} \leftarrow \mathcal{E}_{p.lvl}$;
15. $p \leftarrow Check(p)$;

Locals(in α : *evnt*, p : *node*) : set of *lcl*

Return $\{i \in \mathcal{S}_l : p[i] \neq z_{l-1} \wedge \mathcal{N}_{l,\alpha}(i) \neq \emptyset\}$, the local states in p, a node at level l, that are on a path to $\mathbf{1}$ and locally enable α. In particular, if p is a duplicate of z_l, return $\emptyset$.

Check(in p : *node*) : *node*

If $p[0] = \cdots = p[n_l - 1] = z_{l-1}$, delete p, a node at level l, and return z_l, since $\mathcal{B}(p)$ is $\emptyset$. If p duplicates q in $UT[l]$, delete p and return q. Otherwise, insert p in $UT[l]$ and return p.

Pick(inout $\mathcal{F}$: set of *evnt*) : *evnt*

Remove and return an element from $\mathcal{F}$.

Fire(in α : *evnt*, q : *node*) : *node*

Return s s.t. $\mathcal{B}(s) = \mathcal{N}^*_{\leq q.lvl}(\mathcal{N}_\alpha(\mathcal{B}(q)))$. It is called with $q.lvl < Top(\alpha)$.

1. declare $f,\ s$: *node*;
2. declare $i,\ j$: *lcl*;
3. if $q.lvl < Bot(\alpha)$ then return q;
4. if $Find(FC[q.lvl],(q,\alpha),s)$ then return s;
5. $s \leftarrow NewNode(q.lvl)$;
6. foreach $i \in Locals(\alpha, q)$ do
7. $\quad f \leftarrow Fire(\alpha, q[i])$;
8. $\quad$ foreach $j \in \mathcal{N}_{l,\alpha}(i)$ do
9. $\quad\quad s[j] \leftarrow Union(f, s[j])$;
10. $Saturate(s)$;
11. $Insert(FC[q.lvl], (q, \alpha), s)$;
12. return s;

Union(in p : *node*, q : *node*) : *node*

Assuming that $p.lvl = q.lvl = l$, build an MDD rooted at s encoding $\mathcal{B}(p) \cup \mathcal{B}(q)$, and return s, which is in $UT[l]$.

1. declare i : *lcl*;
2. declare s : *node*;
3. if $p = z_{p.lvl}$ or $p = q$ then return q;
4. if $q = z_{q.lvl}$ then return p;
5. if $Find(UC[p.lvl],\{p,q\},s)$ then return s;
6. $s \leftarrow NewNode(p.lvl)$;
7. for $i = 0$ to $n_{p.lvl} - 1$ do
8. $\quad s[i] \leftarrow Union(p[i], q[i])$;
9. $s \leftarrow Check(s)$;
10. $Insert(UC[p.lvl], \{p, q\}, s)$;
11. return s;

NewNode(in l : *lvl*) : *node*

Create node p at level l with arcs set to z_{l-1}, return p.

Insert(inout *tab*, in *key*, v)

Insert (key, v) in hash table *tab*.

Find(in *tab*, *key*, out v) : *bool*

If (key, x) is in hash table *tab*, set v to x and return *true*. Otherwise, return *false*.

FIGURE 4.10 – Saturation par [CLM07]

160s et 385s pour NuSMV et Cadence SMV respectivement. Les gains en mémoire sont du même ordre : les pics mémoires sont généralement inférieurs en ordre de grandeur par rapport aux deux *model checkers* basés sur SMV. Nous invitons le lecteur intéressé à consulter les tableaux de résultats de [CLM07].

En résumé, la procédure de saturation a pour rôle d'amener les nœuds d'un MDD au plus proche de leurs formes finales, en travaillant du bas vers le haut, le plus rapidement possible. Aussi, cette version de la saturation est dédiée au calcul d'accessibilité.

4.4.2 SDD

La création de nouveaux nœuds SDD est coûteuse à cause de la procédure de canonisation. Des nœuds temporaires inutiles auront donc comme effet d'augmenter le temps de calcul, en plus de participer à l'effet de pic.

Pour illustrer ceci, considérons l'opération $h = (Inc(d,2) + Id)^\star$ où Inc est défini en 4.1. L'application de h sur δ_0 de la figure 4.11 va produire l'application de h sur $a \xrightarrow{\{1\}}$, $b \xrightarrow{\{1\}}$, puis $c \xrightarrow{\{1\}}$ sans entraîner de modifications. La modification ne se fera que sur le dernier nœud, c'est à dire $d \xrightarrow{\{0\}}$. L'application de h n'étant pas terminée, puisqu'il faut atteindre un point fixe, il faut procéder à une nouvelle application de h. Le problème des canonisations inutiles apparaît alors : cette nouvelle application va se faire à nouveau au sommet de δ_1, c'est à dire $a \xrightarrow{\{1\}}$. Retourner à ce sommet implique de devoir canoniser les nœuds a, b et c. Or ils n'ont subi aucune modification, mais ils doivent être stockés indépendamment, car ils sont nouveaux puisqu'ayant des successeurs différents. Finalement, à l'étape (5) de l'évaluation [6], ils n'ont toujours pas été modifiés. Cela signifie donc qu'à chaque itération, il a fallu les canoniser inutilement. Sur la figure 4.11, tous ces nœuds sont encadrés.

La saturation va permettre d'éviter la création de ces nœuds inutiles. [CTM05] adapte la saturation aux SDD. Le principe général reste le même : la localité de chaque évenement de la relation de transition est déterminée. Une table $TopTrans$ référence la plus haute variable que modifie un événement. $TopTrans[i]$ contient un homomorphisme qui est la somme des homomorphismes de toutes les transitions t telles que le plus haut niveau d'application de t est la variable i, et de Id.

Ensuite, un homomorphisme particulier est en charge de parcourir le SDD. À chaque variable il consulte la table $TopTrans$ pour connaître les événements à appliquer en point fixe sur les niveaux inférieurs afin de les saturer.

Dans le cas d'un espace d'états encodé à l'aide d'un SDD hiérarchique, il faut séparer les transitions « locales » des transitions de « synchronisation ». Les premières sont celles qui s'appliquent uniquement à un module encodé par un SDD imbriqué. Les secondes sont celles qui s'appliquent à plusieurs modules. La re-

6. Deux étapes restent en réalité à effectuer pour atteindre le point fixe, mais elles ne changent rien au raisonnement

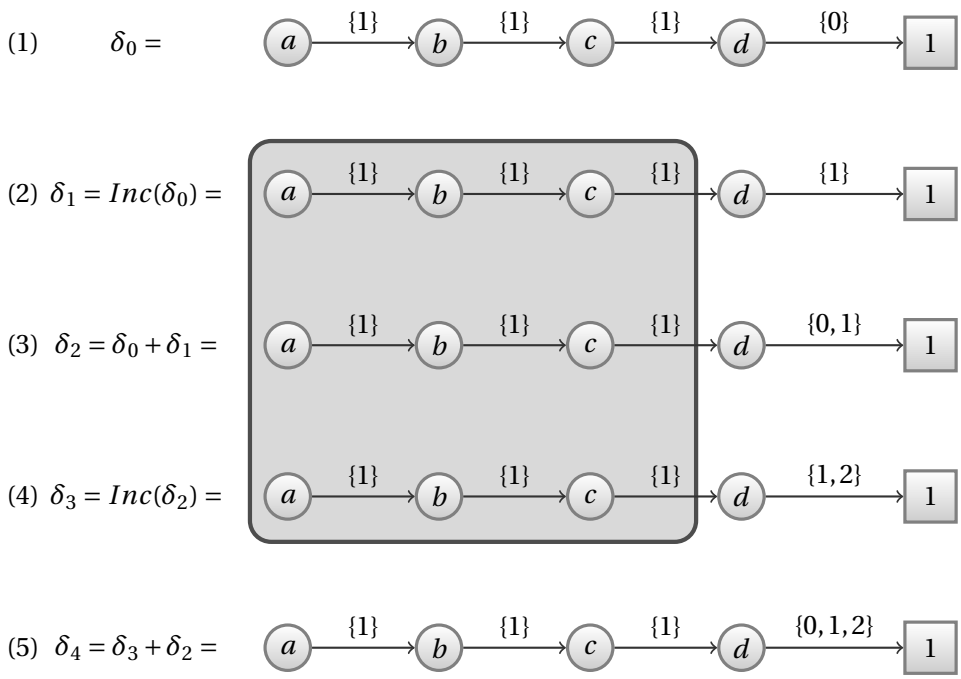


FIGURE 4.11 – Nœuds inutilement créés (encadrés) lors de l'évaluation de $h = (Inc(d,2) + Id)^\star$

lation de transition est ensuite écrite de manière à ce que les transitions locales saturent les composants sur lesquels elles s'appliquent, avant de tirer les transitions de synchronisation qui modifient plusieurs modules à la fois.

Nous reproduisons à la figure 4.12 les performances telles qu'elles apparaissent dans [CTM05]. Ce tableau compare les performances de PNDDD, l'outil implémentant la saturation SDD, à celles de SMART et NuSMV. Les performances sont très bonnes, tout comme SMART, PNDDD gagne en ordre de grandeur sur les résultats de NuSMV.

Proche des SDD, citons [Tov08, HST09] qui implémentent aussi un mécanisme de saturation. Ils utilisent une autre variante des diagrammes de décision, les *Interval Decision Diagrams* [ST98]. Ces derniers sont assez proches des SDD par le fait qu'ils encodent des ensembles de valeurs sur les arcs. Toutefois, ils ne supportent pas la hiérarchie car le domaine des variables est limité à $\mathbb{N}$. Le principe général reste le même : tant qu'il est possible de faire évoluer un niveau d'un diagramme de décision, les opérations s'y appliquant sont répétées jusqu'à obtenir un point fixe. À l'inverse des SDD, cette variante utilise des diagrammes de décision pour encoder les relations de franchissement.

Model	N (#)	States (#)	final SDD (#)	final DDD (#)	total SDD (#)	total DDD (#)	PNDDD SDD time (sec)	NuSMV time (sec)	SMaRT time (sec)
Philosophers	100	4.97+62	398	21	2185	70	0.21	990.8	0.43
	200	2.47+125	798	21	4385	70	0.43	18129	0.7
	1000	9.18e+626	3998	21	21985	70	2.28	-	5.9
	5000	6.52+3134	19998	21	109985	70	11.7	-	83.7
Ring	10	8.29e+09	61	44	2640	150	0.4	6.1	0.11
	15	1.65e+16	288	44	8011	150	1.21	2853	0.29
	50	1.72e+52	2600	44	238400	150	34.01	-	5.6
FMS	25	8.54e+13	55	412	346	11550	0.26	41.6	0.36
	50	4.24e+17	105	812	671	38100	1.02	17321	1.33
	80	1.58e+20	165	1292	1064	89760	2.59	-	4
	150	4.8e+23	305	2412	1971	294300	10.52	-	20.7
Kanban	10	1.01e+09	15	46	129	592	0.02	?	0.48
	50	1.04+16	55	206	1589	9972	1.08	?	43
	100	1.73e+19	105	406	5664	37447	8.79	?	474
	200	3.17e+22	205	806	21314	144897	93.63	?	13920
Lotos [8]	N/A	9.79474e+21	326	759	125773	34298	265.28	?	?
AGV [14]	N/A	3.09658e+07	12	34	135	234	0.01	?	?
AGV Controlled	N/A	1.66011e+07	95	124	2678	349	0.38	?	?

FIGURE 4.12 – Performances de PNDDD

4.5 Problématiques

La saturation est certainement à l'heure actuelle la technique de franchissement des transitions d'un système la plus efficace. Elle apporte des gains de temps et de mémoire possiblement exponentiels. Cela est dû au fait qu'elle évite la création de nœuds temporaires qui ne seront pas utiles au diagramme de décision final.

Pourtant, les méthodes présentées ne sont pas sans inconvénients. [CMS03] nécessite de connaitre les bornes des systèmes vérifiés ou alors utilisent des techniques complexes de mise à jour des matrices représentant la relation de franchissement. De plus, les états locaux potentiels sont représentés explicitement dans une structure externe au MDD. Cela peut poser problème s'il y a trop d'états locaux pour un sous-système. Les auteurs précisent d'ailleurs à ce propos qu'il est nécessaire de découper plus finement le système le cas échéant.

De plus, cette méthode n'est pas capable de discerner au sein d'une transition des sous-éléments pouvant eux-aussi participer à la saturation. Par exemple, des compositions de transitions seront traitées d'un seul bloc, empêchant d'appliquer plus efficacement chaque opérande.

Quant à [CTM05], son principal inconvénient est que les homomorphismes permettant la saturation sont écrits spécifiquement pour les réseaux de Petri. Gérer un nouveau formalisme nécessite donc de développer de nouveaux homomorphismes, pensés pour créer un effet de saturation. Ce n'est pas une tâche aisée car comprendre en détail la saturation nécessite de maîtriser complètement les dia-

grammes de décision sous-jacents.

Un inconvénient commun à ces deux variantes est qu'elles ne s'appliquent que sur une seule fermeture transitive extérieure permettant de traiter l'accessibilité. Or la vérification de formules de logiques temporelles nécessite une série de point fixes imbriqués.

De plus, l'utilisation des diagrammes de décision ne se limite pas au *model checking*. Or les solutions de la littérature présentées ici se cantonnent à cette problématique. Les homomorphismes offrent déjà un mécanisme général de manipulation des diagrammes de décision. Il faut donc en tirer partie afin d'être en mesure de proposer ce phénomène de saturation à tout type d'utilisation.

Nous proposons dans le chapitre suivant des méthodes permettant de généraliser la saturation à certaines formes de fermeture transitive, exprimées à l'aide d'homomorphismes. Nous montrerons aussi comment nous automatisons la saturation sans imposer aux utilisateurs quoi que ce soit qu'ils ne connaissent déjà pour écrire les opérations adaptées à leurs formalismes, tout en éliminant les inconvénients énoncés ci-dessus.

CHAPITRE 5

Saturation automatique

Dans ce chapitre, nous présentons la deuxième contribution majeure de ces travaux qui est la **saturation automatique** sur les SDD. Deux publications ont diffusé en partie cette contribution : [HTMK08] et [HTMK09].

La saturation présentée à la section 4.4 page 82 permet des gains possiblement exponentiels aussi bien en temps qu'en espace mémoire. Cependant, sa mise en œuvre requiert des connaissances très avancées sur les diagrammes de décision manipulés. Écrire la relation de transition correspondant à un formalisme et de l'adapter aux diagrammes de décision étant déjà complexe, on ne peut espérer demander au concepteur d'un formalisme de prendre également en compte ce phénomène de saturation. De ce fait, il faut pouvoir fournir aux utilisateurs de diagrammes de décision un mécanisme permettant d'automatiser le processus de saturation.

L'idée générale de la saturation est d'appliquer les parties locales d'une relation de transition à un nœud d'un diagramme de décision jusqu'à atteindre un point fixe. Tant qu'un nœud se voit modifié, l'évaluation est propagée vers le bas du diagramme de décision. Lorsqu'un point fixe est atteint, l'évaluation remonte au nœud supérieur.

La saturation repose sur deux idées :

– le point fixe ;

– la localité d'une opération.

L'identification de ces principes dans une relation de transition serait ainsi à même de fournir les informations nécessaires pour en automatiser la transformation afin d'obtenir cet effet de saturation.

La structuration des homomorphismes, c'est à dire un ensemble d'opérations (homomorphismes inductifs) liées entre elles par des opérateurs (union, intersection, etc.) va permettre de détecter des motifs que l'on pourra transformer afin de produire un effet de saturation.

Le travail présenté ici manipule l'arbre de syntaxe abstraite (*Abstract Syntax*

Tree : *AST*) des homomorphismes que l'utilisateur a fourni afin d'en produire un nouveau, à la sémantique équivalente, permettant la saturation.

Ce chapitre décrit ces identifications et l'utilisation des informations obtenues dans le but d'atteindre la saturation automatique. De plus, nous généralisons la saturation en utilisant des informations que les techniques de la littérature n'exploitaient pas.

Produire la saturation automatique va se faire en plusieurs étapes :

1. Déterminer la localité des homomorphismes (section 5.1)
2. Identifier des homomorphismes remarquables (section 5.2)
3. Combiner ces informations afin de procéder à une réécriture de l'AST (section 5.3)

La section 5.4 présente les résultats obtenus, puis la section 5.5 présente un formalisme de modélisation créé pour tirer parti des SDD. Enfin, nous concluons sur la saturation automatique en section 5.6

5.1 Propagation et domaines d'application des homomorphismes

Comme vu en section 4.4 page 82, de nombreuses applications d'homomorphismes sont faites inutilement. En connaissant les variables sur lesquelles ces opérations s'appliquent, il est possible de les éviter, permettant ainsi de limiter le nombre de création de nouveaux nœuds.

À cette fin, nous introduisons deux mécanismes d'identification des variables touchées par une opération. Le premier, section 5.1.1, travaille de manière dynamique à l'évaluation tandis que le deuxième, section 5.1.2, travaille de manière statique et est utilisé à la construction de l'AST.

Ensuite, nous définissons une classe particulière des homomorphismes, les sélecteurs, en section 5.1.3.

5.1.1 Invariance locale et propagation

Nous définissons ici la brique de base rendant la saturation automatique possible. Le but de la saturation étant d'éviter des applications inutiles, nous avons besoin de connaître la plus haute variable sur laquelle s'applique une opération. Dans ce but, nous définissons les homomorphismes **localement invariants** :

Définition 5.1 : Homomorphisme localement invariant
Un homomorphisme h est localement invariant à une variable ω si et seulement si :

$$\forall \delta = \langle \omega, \pi, \alpha \rangle \in \mathbb{S}, h(\delta) = \sum_{\langle s, \delta' \rangle \in \alpha} \omega \xrightarrow{s} h(\delta')$$

Concrètement, cela signifie que l'application de h ne modifie pas la structure des nœuds de variable ω, et que h n'est pas modifié lorsqu'il parcourt ces nœuds. La variable ω n'est donc jamais lue ou écrite par h. Il est à noter que l'homomorphisme Id est localement invariant quelle que soit la variable. Pour un homomorphisme inductif h localement invariant à ω, cela signifie que $h(\omega \xrightarrow{s}) = \omega \xrightarrow{s} h$. Il est donc possible de *propager* l'homomorphisme jusqu'à son site d'application le plus haut dans le SDD (figure 5.1).

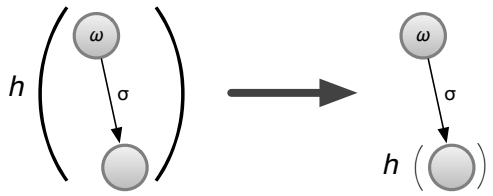


FIGURE 5.1 – Propagation de l'homomorphisme h, localement invariant à ω

Pour connaître cette information d'invariance locale, les homomorphismes doivent fournir le prédicat $\mathsf{V}(h, \omega)$. Celui-ci est vérifié si h est localement invariant à ω. Cette information minimale sera utilisée pour réordonner l'application des homomorphismes pour produire la saturation.

Il n'est pas difficile pour un concepteur d'homomorphisme inductif de fournir ce prédicat, puisqu'en réalité il connaît déjà les variables sur lesquelles ils s'appliquent. Cela simplifie même l'écriture des homomorphismes, puisque la propagation est séparée des applications réellement effectuées.

Remarque
Par défaut, si V n'est pas indiqué dans la définition d'un homomorphisme, alors ce dernier n'est jamais localement invariant.

Par exemple, considérons les homomorphismes h^- (homomorphisme 4.3 page 80) et h^+ (homomorphisme 4.4 page 81). En utilisant le prédicat V, nous pouvons les réécrire de la manière suivante :

Homomorphisme 5.1 : h^-

$$\begin{array}{ll} h^-(p,m)(\omega \xrightarrow{s}) & = \omega \xrightarrow{\{n-m \mid n \in s \wedge n \geq m\}} Id \\ h^-(p,m)(\boxed{1}) & = \boxed{0} \end{array}$$

$$\mathsf{V}(h^-,\omega) = \qquad \omega \neq p$$

Homomorphisme 5.2 : h^+

$$\begin{array}{ll} h^+(p,m)(\omega \xrightarrow{s}) & = \omega \xrightarrow{\{n+m \mid n \in s\}} Id \\ h^+(p,m)(\boxed{1}) & = \boxed{0} \end{array}$$

$$\mathsf{V}(h^+,\omega) = \qquad \omega \neq p$$

L'application d'un homomorphisme inductif ϕ à $\delta = \langle \omega, s, \alpha \rangle$ est normalement définie par $\phi(\delta) = \sum_{\langle s,\delta' \rangle \in \alpha} \Phi(\omega \xrightarrow{s})(\delta')$. Mais quand Φ est localement invariant à ω, le calcul de cette union devient $\sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} \Phi(\delta')$. Cela revient donc à **propager** l'évaluation d'un homomorphisme sur les successeurs d'un nœud pour lequel l'homomorphisme est localement invariant.

Tous les homomorphismes « opérateurs » (union, composition, etc.) sont quant à eux localement invariants à une variable ω si toutes leurs opérandes sont eux aussi localement invariants à ω. Par exemple, pour la composition $h = h_1 \circ h_2$, $\mathsf{V}(h,\omega) \Leftrightarrow \mathsf{V}(h_1,\omega) \wedge \mathsf{V}(h_2,\omega)$.

Le fait que ces homomorphismes soient locaux en fonction de leurs opérandes dérive de la définition des operations ensemblistes sur les SDD. Par exemple, dans le cas de l'union et de deux homomorphismes h et h' localement invariants à la variable ω, nous avons : $\forall \delta = \langle \omega, \pi, \alpha \rangle \in \mathbb{S}$,

$$\begin{aligned} (h+h')(\delta) &= h(\delta) + h'(\delta) \\ (1) &= \sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} h(\delta') + \sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} h'(\delta') \\ (2) &= \sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} h(\delta') + h'(\delta') \\ (3) &= \sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} (h+h')(\delta') \end{aligned}$$

Le passage de (1) à (2) correspond à la canonisation des SDD. Celui de (2) à (3) correspond à la propriété de linéarité.

Ceci permet aux homomorphismes opérandes de ces opérateurs de descendre en un seul groupe tant qu'ils sont localement invariants. Dès qu'ils ne se propagent plus, alors le mécanisme d'évaluation standard est utilisé. La descente groupée implique de meilleurs temps et une moindre consommation mémoire puisque des applications inutiles sont évitées.

D'un point de vue technique, l'invariance locale nous permet de créer un nouveau nœud en copiant directement la structure du nœud original et de modifier cette copie. En effet, l'application de ϕ enlèvera au pire quelques arcs. Si $\phi(\delta')$ produit le terminal $\boxed{0}$, nous supprimons l'arc y menant. Autrement, si deux évaluations $\phi(\delta')$ et $\phi(\delta'')$ retournent le même résultat, nous devons fusionner les deux arcs y menant en un arc étiqueté par l'union de leurs valeurs.

Notons que si la propagation d'un homomorphisme ne permet pas, à elle seule, d'obtenir l'effet de saturation, elle engendre mécaniquement une optimisation non négligeable. En effet, comme cela a été vu à la section 4.4 page 82, appliquer une opération à un nœud implique une procédure de canonisation coûteuse. De ce fait, même si une opération ne modifie pas le niveau courant, son application aux niveaux inférieurs oblige à canoniser le niveau courant. Le fait de propager une opération jusqu'à son niveau le plus haut d'application permet de n'effectuer cette canonisation que lorsque l'évaluation de l'homomorphisme sera terminée et revenue à son plus haut niveau d'application.

Un autre avantage est une meilleure utilisation du cache d'applications des homomorphismes. En effet, celui-ci est sollicité à chaque application et une entrée sera créée pour chaque nouvelle application. Donc le fait de ne pas appliquer une opération à un niveau auquel elle n'apporterait aucune modification permet d'éviter ces entrées inutiles dans le cache. De plus, la propagation par blocs implique qu'une seule entrée dans le cache représentera les homomorphismes groupés. D'un point de vue pragmatique, cela évite une utilisation inutile de la mémoire et entraîne donc moins de procédures de récupérations de mémoire à l'aide d'un ramasse-miettes, ce qui amène inévitablement un gain en temps.

Ce prédicat est dynamiquement évalué lors de l'application d'un homomorphisme. Cela est nécessaire car les SDD peuvent contenir des séquences de longueur variable dont la taille évolue au cours du temps.

5.1.2 Domaine d'application

Si l'invariance locale est très utile à certaines réécritures permettant la saturation automatique, elle ne permet pas de connaître la commutativité de deux opérations. Pour connaître cette information , il est nécessaire d'avoir connaissance de l'ensemble des variables sur lesquelles s'appliquent ces opérations, au moment de leurs constructions. Or l'invariance locale n'est évaluée que lors de la descente des homomorphismes sur les SDD manipulés.

Si jamais deux opérations n'ont aucune variable en commun, alors il est pos-

sible d'inverser leurs évaluations. Nous introduisons à cette fin le domaine d'application d'un homomorphisme :

Définition 5.2 : Domaine d'application
Le domaine d'application d'un homomorphisme h, noté $\mathsf{D}(h)$, est l'ensemble des variables $\omega_i \in Var$ tel que

$$\mathsf{D}(h) = Var \setminus \{\omega_i | \mathsf{V}(h, \omega_i)\}$$

Remarque
Le domaine d'application par défaut d'un homomorphisme est Var, l'ensemble des variables. Nous l'indiquerons dans les définitions d'homomorphismes seulement si ces derniers s'appliquent sur un sous-ensemble strict de Var.

Cette définition plus générale de l'invariance locale, puisque définie pour toutes les variables, nous offre plus d'informations car elle permet de comparer deux homomorphismes par leurs domaines. Cela nous permet de déterminer si deux homomorphismes sont indépendants l'un de l'autre.

De même que pour l'invariance locale, chaque homomorphisme opérateur possède le même domaine d'application que ses opérandes.

À la différence de l'invariance locale seule, cette information est demandée aux homomorphismes à leurs constructions. Elle est donc purement statique. Dans le cas de SDD à structure contante, c'est à dire sans séquence de longueur variable, les informations fournies par l'une et par l'autre sont strictement identiques (puisque V énumérera toutes les variables non touchées à l'évaluation). Par contre, s'il existe des séquences de longueur variable, alors le domaine d'application sera une sur-approximation. Un homomorphisme inductif devra en effet fournir comme domaine d'application l'intervalle allant de la variable la plus haute qu'il modifie jusqu'au terminal $\boxed{1}$, puisqu'il lui est impossible de connaître les variables qui seront possiblement touchées par les homomorphismes qu'il pourrait générer.

L'indépendance de deux homomorphismes nous permettra de les réordonner au sein des compositions afin de rapprocher les opérations s'appliquant sur les mêmes variables (section 5.3.2)).

5.1.3 Homomorphisme sélecteur

L'expérience aidant, nous nous sommes aperçus que bien souvent des homomorphismes étaient écris uniquement à la seule fin de « sélectionner » des che-

mins. Ou encore, d'autres effectuent un filtrage sur les valeurs portées avant de les modifier. Dans ce dernier cas, il serait possible d'écrire ces homomorphismes en deux opérations : l'une effectuant le filtre, et agissant ainsi en tant que sélecteur, l'autre procédant à la transformation des valeurs filtrées.

Les formules de la logique temporelle CTL en sont un bon exemple, puisqu'elles ont pour rôle de filtrer les états jusqu'à obtenir ceux respectant un ensemble de propriétés recherchées.

Il est donc devenu nécessaire de rendre disponible un mécanisme permettant aux concepteurs d'homomorphismes d'identifier clairement cette particularité. Cette propriété permettra de détecter des motifs pour lesquels une optimisation est possible.

Nous introduisons donc le prédicat S indiquant qu'un homomorphisme est un sélecteur. Un tel homomorphisme est défini de la manière suivante :

Définition 5.3 : Sélecteur
Un homomorphisme h est un sélecteur si et seulement si

$$\forall \delta \in \mathbb{S}, h(\delta) \subseteq \delta$$

Autrement dit, un homomorphisme vérifie S s'il ne renvoie qu'un sous-ensemble des ensembles rencontrés sur tous les chemins d'un SDD.

Remarque
Le prédicat S sera indiqué dans la définition d'un homomorphisme seulement si ce dernier est un sélecteur.

À titre d'exemple, considérons l'homomorphisme Inc défini en 4.1 page 75. Son rôle est d'incrémenter toutes les valeurs d'une variable d'un chemin, et ce tant qu'elles sont inférieures à une certaine valeur. Nous pouvons donc réécrire Inc en $Inc' \circ Leq$ où Inc' ne fait qu'incrémenter les valeurs et Leq ne fait que sélectionner les valeurs inférieures à un seuil :

Homomorphisme 5.3 : Inc'

$$\begin{aligned} Inc'(\omega \xrightarrow{s}) &= \omega \xrightarrow{\{x+1 | x \in s\}} Inc' \\ Inc'(\boxed{1}) &= \boxed{1} \end{aligned}$$

Homomorphisme 5.4 : Leq

$$\begin{array}{ll} Leq(v,n)(\omega \xrightarrow{s}) & = \omega \xrightarrow{\{x|x\in s \wedge x<n\}} Leq \\ Leq(v,n)(\boxed{1}) & = \boxed{1} \end{array}$$

$$\begin{array}{ll} \mathsf{D}(Leq) = & \{v\} \\ \mathsf{S}(Leq) = & \top \end{array}$$

Les sélecteurs offrent plusieurs avantages. Considérons un homomorphisme sélecteur h et g un homomorphisme quelconque , $\forall \delta \in \mathbb{S}$:

$$(1)\ h^\star(\delta) = h(\delta)$$
$$(2)\ (h + Id)(\delta) = Id(\delta)$$

Le point fixe sur un sélecteur (1) ne présente aucun intérêt car une fois que ce dernier a sélectionné un certain nombre de chemins, il n'en retirera pas plus à l'itération suivante. De même, sélectionner un ensemble de chemin de δ pour ensuite l'y ajouter produit à nouveau δ (2).

Il est possible de réduire le coût de création d'un nœud engendré par l'évaluation d'un sélecteur. En effet, comme ce type d'homomorphisme retourne un sous-ensemble, la partition des valeurs sur les arcs n'est pas modifiée, seuls peuvent changer les successeurs. Il n'y a donc pas besoin dans ce cas de procéder à l'union de tous les nouveaux chemins calculés : il suffit de vérifier qu'il n'existe pas deux arcs menant au même successeur. Le cas échéant, on fusionne ces arcs (*cf.* figure 4.2 page 74) [1].

À l'heure actuelle, il nous est impossible de déterminer automatiquement si un homomorphisme est un sélecteur. Il n'est absolument pas envisageable de détecter à l'exécution si cette condition est respectée. Il faudrait pour cela, à chaque évaluation d'homomorphisme, tester si les ensembles retournés sont inclus dans ceux de départ, ce qui est totalement inefficace. Il serait aussi possible de détecter cette information à la création des homomorphisme. Mais cela nécessite soit de faire de l'introspection sur le code, ce qui n'est pas de notre ressort, soit d'utiliser un langage de haut-niveau permettant d'écrire des homomorphismes. Ce dernier cas a été envisagé, mais l'apport a été jugé insuffisant : il ne s'agirait que d'une contribution technique qui n'apporte rien à la saturation automatique.

1. Il est possible d'effectuer la même optimisation pour tout homomorphisme qui serait une application injective (la partition resterait en effet cohérente), mais nous n'avons pas encore utilisé ce principe.

Pour toutes ces raisons, nous laissons pour l'instant le soin à l'utilisateur de nous préciser le statut de ses homomorphismes. De plus, connaître ou non cette information n'empêche pas la saturation de s'activer. Tout au plus certains motifs ne seront pas détectés et donc non optimisés. Enfin, cela reste une information que maîtrise l'utilisateur.

5.2 Homomorphismes remarquables

Dans le section précédente, nous avons mis en place deux manières d'identifier les variables sur lesquelles s'appliquent des opérations, statiquement ou dynamiquement. Nous identifions aussi les homomorphismes sélecteurs qui ne renvoient que des sous-ensembles.

Nous introduisons des nouveaux types d'homomorphismes qui enrichissent le cadre de travail des SDD de concepts de haut-niveau, facilitant la conception d'opérations aux utilisateurs. Ces homomorphismes sont le fruit de l'expérience que nous avons retirée lors de leurs utilisations et que nous avons identifiées comme étant des « bonnes pratiques ».

De plus, grâce à l'identification de ces nouveaux homomorphismes, des règles de réécritures sémantiques pourront s'appliquer pour produire un effet de saturation.

5.2.1 Structure conditionnelle et complément

Le prédicat S nous permet de créer deux nouveaux homomorphismes : un permettant de décrire une structure conditionnelle et un calculant le complément d'un sélecteur.

Il est possible de calculer le complément des sélecteurs car ceux-ci ne renvoient qu'un sous-ensemble. Il n'est pas possible de calculer ce complément dans le cas général car les SDD ont possiblement un domaine infini sur les variables. Le complément d'un homomorphisme est défini comme suit :

Définition 5.4 : Complément

$$\overline{h}(\delta) = Id(\delta) - h(\delta)$$
où $\mathsf{S}(h)$ est vérifié

Le calcul du complément nous permet d'introduire un nouvel homomorphisme opérateur qui simule une structure conditionnelle. Cet homomorphisme *IfThenElse* (ITE) est défini de la manière suivante :

Définition 5.5 : Structure conditionnelle

$$ITE(test, g, h)(\delta) = (g \circ test + h \circ \overline{test})(\delta))$$
où $\mathsf{S}(test)$ est vérifié

Cet homomorphisme partitionne δ grâce au sélecteur $test$ et applique g sur les chemins vérifiant $test$, et applique h sur le complément. Il permet ainsi d'aisément appliquer un traitement sur des parties complémentaires.

Notons que si l'utilisateur voit cet homomorphisme d'un seul tenant, nous le laissons sous la forme d'une somme de deux compositions. En effet, cela nous permet d'être en mesure d'identifier les compositions que nous pouvons transformer afin de les rendre plus efficaces.

5.2.2 Application locale

L'introduction des SDD a apporté la hiérarchie aux diagrammes de décision. Dans ce contexte, il faut pouvoir produire la saturation même sur les SDD imbriqués dans une hiérarchie. Autrement dit, l'application d'un homomorphisme sur un SDD imbriqué doit ne remonter au niveau hiérarchique supérieur que lorsque cette application imbriquée a atteint un point fixe.

Homomorphisme 5.5 : Application locale manuelle d'un homomorphisme

$$\begin{aligned} g(v,h)(\omega \xrightarrow{s}) &= \omega \xrightarrow{h(s)} Id \\ g(v,h)(\boxed{1}) &= \boxed{0} \end{aligned}$$

$$\begin{aligned} \mathsf{V}(g,\omega) &= \omega \neq v \\ \mathsf{D}(g) &= \{v\} \\ \mathsf{S}(g) &= \mathsf{S}(h) \end{aligned}$$

Jusqu'à présent, appliquer une opération sur une sous-hiérarchie imposait à l'utilisateur d'écrire un homomorphisme inductif procédant lui-même à cette application, comme le montre l'exemple 5.5 dans lequel g applique h dès lors qu'il rencontre la variable v. De ce fait, l'analyse de l'AST ne montrerait qu'un seul homomorphisme, pour lequel la seule information disponible est son domaine d'application. On ne pourra donc pas savoir s'il effectue un quelconque travail en hiérarchie. La figure 5.2 illustre ceci pour l'opération $Id + g(v,h)$ où g est l'homomorphisme 5.5.

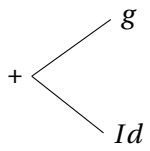


FIGURE 5.2 – AST de $Id + g(v,h)$

Dans le but d'avoir un plus grand contrôle, nous introduisons l'homomorphisme $\mathscr{L}$ défini comme suit :

Définition 5.6 : Application locale d'un homomorphisme

$$\begin{array}{ll} \mathscr{L}(v,h)(\omega \xrightarrow{s}) & = \omega \xrightarrow{h(s)} Id \\ \mathscr{L}(v,h)(\boxed{1}) & = \boxed{0} \end{array}$$

$$\begin{array}{rl} \mathsf{V}(\mathscr{L},\omega) = & \omega \neq v \\ \mathsf{D}(\mathscr{L}) = & \{v\} \\ \mathsf{S}(\mathscr{L}) = & \mathsf{S}(h) \end{array}$$

L'homomorphisme $\mathscr{L}$ est paramétré par v, la variable du diagramme de décision qui contient le SDD imbriqué sur lequel il faut appliquer h.

Il agit comme un « transporteur » d'homomorphisme. En effet, il se propage ($\omega \xrightarrow{s} \mathscr{L}(v,h)$) tant qu'il ne rencontre pas la variable v cible. Lorsque cette variable est rencontrée, alors il applique h sur l'ensemble s situé à ce niveau ($\omega \xrightarrow{h(s)} Id$). $\mathscr{L}$ est donc localement invariant à toutes les variables autres que v.

Pour illustrer ce nouvel homomorphisme, nous utilisons Inc', défini en 5.3 : l'application de $\mathscr{L}(y, Inc')$ sur $x \xrightarrow{a \xrightarrow{\{0,1\}} \boxed{1}} y \xrightarrow{a \xrightarrow{\{0,1\}} \boxed{1}} \boxed{1}$ aura ainsi pour résultat $x \xrightarrow{a \xrightarrow{\{0,1\}} \boxed{1}} y \xrightarrow{a \xrightarrow{\{1,2\}} \boxed{1}} \boxed{1}$. Seul le SDD imbriqué au niveau de la variable y a été touché par l'opération.

$\mathscr{L}$ répond à un besoin qui s'est exprimé de nombreuse fois : toutes nos applications des SDD font usage de cet homomorphisme.

Dorénavant, détecter une application en hiérarchie revient à détecter la présence de $\mathscr{L}$. Sur l'AST de la figure 5.3, nous voyons que toutes les opérations cachées précédemment par l'homomorphisme inductif de l'utilisateur sont visibles et donc disponibles pour une éventuelle transformation.

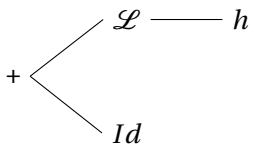


FIGURE 5.3 – AST de $Id + \mathscr{L}(v,h)$

L'introduction de $\mathscr{L}$ a un double intérêt : non seulement il permet la détection d'une application en hiérarchie, mais il permet également de faciliter la tâche à l'utilisateur en lui offrant le moyen de séparer clairement le mécanisme d'application sur une hiérarchie de sa propre relation de transition.

5.2.3 Composition commutative

La composition est au cœur de l'usage des SDD : elle est très souvent utilisée dans l'écriture d'une relation de transition. Par exemple, si l'on considère les réseaux de Petri, l'opération représentant une transition est formée par la composition de tous les homomorphismes représentant les arcs Pré et Post.

La composition impose un ordre d'application : $a \circ b$ nécessite d'évaluer b avant a. Or, bien souvent dans une relation de transition, il s'avère que les opérandes sont commutatifs et de ce fait le résultat sera correct quelque soit l'ordre d'évaluation.

À tire d'exemple, considérons la composition des homomorphismes $h^-(a,1)$ (homomorphisme 5.1 paramétré par a et 1) et $h^+(b,1)$ (homomorphisme 5.2 paramétré par b et 1) s'appliquant sur $a \xrightarrow{0} b \xrightarrow{1} \boxed{1}$. Appliquer $h^-(a,1)$ en premier renvoie immédiatement le terminal $\boxed{0}$, $h^+(b,1)$ ne sera donc pas évalué dans ce cas. À l'inverse, appliquer $h^+(b,1)$ en premier produit $a \xrightarrow{0} b \xrightarrow{2} \boxed{1}$, puis $h^-(a,1)$ renvoie aussi le terminal $\boxed{0}$.

Nous introduisons la composition commmutative comme suit :

Définition 5.7 : Composition commutative
$a \circ b$ est une composition commutative si et seulement si $a \circ b = b \circ a$.
On notera $a \oplus b$ une telle composition.

De par son caractère commutatif, il est possible de considérer cette forme de composition comme étant une opération n-aire. Cela permet d'évaluer les opérandes avec plus de latitude. Par exemple, cela nous permet de fusionner des applications locales à une même variable. Cette souplesse dans l'ordre d'évaluation est exploitée pour la saturation.

Dans le cadre des SDD, déterminer la commutativité de deux opérations revient à déterminer si le résultat de l'une influe sur l'autre. On ne veut pas imposer

à l'utilisateur d'être en mesure de déterminer si ses opérations sont commutatives avec toutes les autres possiblement existantes. Une approximation conservative est donc calculée par la fonction $commutatives$ définit ci-après :

Définition 5.8 : Test de la commutativité
$arecom(h,h') : \boldsymbol{H} \times \boldsymbol{H} \rightarrow \mathbb{B}$ est définie par :

$$\begin{aligned} & \mathsf{D}(h) \cap \mathsf{D}(h') = \varnothing \\ \vee\ & \mathsf{S}(h) \wedge \mathsf{S}(h') \\ \vee\ & h = \mathscr{L}(v,g) \wedge h' = \mathscr{L}(v,g') \wedge commutatives(g,g') \end{aligned}$$

Deux homomorphismes h et h' sont donc commutatifs s'ils ne s'appliquent pas sur les mêmes variables. Cela se détecte facilement si leurs domaines d'applications sont disjoints.

Lorsqu'il s'agit de sélecteurs, ceux-ci retournant uniquement des sous-ensembles des valeurs d'entrées (sans les modifier), alors l'application de l'un à l'autre consiste en une intersection des valeurs qu'ils reconnaissent. L'intersection étant une opération commutative, il est possible d'intervertir l'application des deux sélecteurs.

Quand il s'agit de la composition de deux applications locales ciblant la même variable v, il est possible de les faire commuter si les homomorphismes qui sont appliqués sur la hiérarchie imbriquée sont eux même commutatifs.

Cette fonction peut être invoquée à la construction des homomorphismes car les informations qu'elle requiert ne sont pas dépendantes de l'évaluation.

5.3 Réécriture des homomorphismes

Une fois réunies les informations de localité et de types d'homomorphismes remarquables, il est possible de procéder à une série de réécritures sur l'AST des opérations. Certaines de ces réécritures sont faites statiquement à la construction des homomorphismes, alors que d'autres sont exécutées dynamiquement au cours de l'évaluation.

Les réécritures statiques correspondent aux cas où il suffit de changer la structure des opérations, mais pas leurs évaluations. Par exemple, la transformation d'une succession d'unions binaires en une union n-aire ne nécessite pas de modification dans le mécanisme d'évaluation. Les réécritures dynamiques sont effectuées lorsqu'il est nécessaire de modifier la procédure d'évaluation. Ce sont en effet des transformations qui se basent sur le prédicat V qui indique l'invariance locale. Or ce prédicat ne peut être consulté que lorsqu'on a connaissance de la variable courante sur laquelle on s'applique.

5.3.1 Union

Nous procédons à deux transformations sur l'union. La première consiste simplement à réunir les opérandes d'union binaires consécutives, lors de leurs constructions, pour créer une seule opération n-aire. De cette manière nous créons un « paquet » d'homomorphismes qu'il sera possible de propager sur un SDD en une seule fois.

La deuxième est effectuée à l'évaluation : lors de l'application d'une union n-aire $H = \sum_i h_i$ sur un nœud $\delta = \langle \omega, \pi, \alpha \rangle$, nous créons une partition sur ses opérandes :

$$H = F \cup G \text{ avec } F = \{h_i | \forall (h_i, \omega)\} \text{ et } G = \{h_i | \neg \forall (h_i, \omega)\}$$

L'application $H(\delta)$ est alors écrite :

$$H(\delta) = (\sum_{f \in F} f)(\delta) + (\sum_{g \in G} g)(\delta)$$
$$\text{qu'on notera } F(\delta) + G(\delta)$$

La partie F correspond ainsi à l'ensemble des opérandes de H localement invariants à ω et qui peuvent donc être propagés ensemble. La partie G est quand à elle évaluée de manière classique ($G(\delta) = \sum_{g \in G} g(\delta)$). La figure 5.4 montre cette nouvelle évaluation.

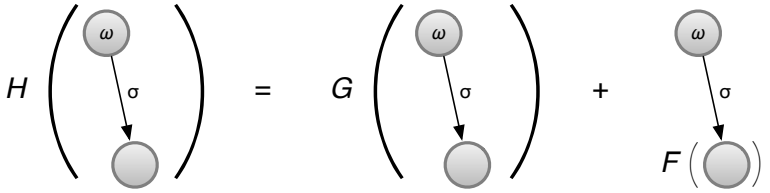


FIGURE 5.4 – Réécriture de l'union $H = F + G$

L'application classique $H(\delta) = \sum_i h_i(\delta)$ implique un parcours indépendant de chaque h_i sur δ et donc force la canonisation de chaque $h_i(\delta)$. Cette transformation implique au contraire que ce parcours est effectué d'un seul bloc pour tous les homomorphismes localement invariants à la variable courante. Le bloc ainsi propagé sera adapté au fur et à mesure de la descente, puisque la partition se fait pour chaque variable.

Dans la pratique, le calcul de cette partition n'est effectué qu'une seule fois par variable, à la première rencontre, et non pas à chaque fois qu'une union s'applique sur une variable, car son résultat est rangé dans un cache. Il en est d'ailleurs ainsi pour toutes les réécritures dynamiques.

5.3.2 Composition

Cette transformation consiste à remplacer, si possible, une composition de deux opérations commutatives par une composition commutative. S'appuyant sur la fonction $commutatives$, que l'on peut invoquer statiquement, elle est effectuée à la construction des compositions.

Nous pouvons écrire, lorsque $commutatives(h, h')$ est vraie :

$$h \circ h' = h \oplus h'$$

La composition étant une opération n-aire, il est possible d'agréger les opérandes de compositions commutatives binaires successives pour ne former qu'une composition commutative n-aire. Toutefois, $h_1 \oplus h_2$ et $h_2 \oplus h_3$ n'implique pas $commutatives(h_1, h_3)$, puisque h_1 et h_2 peuvent travailler sur les mêmes variables. Nous allons donc procéder de manière plus avancée à la construction d'une composition commutative. Soit $h = \bigoplus_i h_i$ une composition commutative et h' un homomorphisme quelconque :

$$C = \{h_i | commutatives(h_i, h')\}$$
$$\overline{C} = \{h_i | \neg commutatives(h_i, h')\}$$
$$h \circ h' = \begin{cases} \text{si } C = \emptyset : h \circ h' \\ \text{si } \overline{C} = \emptyset : \left(\bigoplus_i h_i\right) \oplus h' \\ \text{sinon} : \bigoplus_{i \in C} h_i \oplus \left(\left(\bigoplus_{i \in \overline{C}} h_i\right) \circ h'\right) \end{cases}$$

Pour ajouter un homomorphisme h' quelconque dans une composition commutative, on commence par créer deux parties C et $\overline{C}$ contenant respectivement les opérandes de h qui commutent avec h' et ceux qui ne commutent pas. Si jamais aucun opérande ne commute avec h', alors il faut respecter la séquentialité des opérations h et h'. Dans le cas contraire, il suffit d'insérer h' dans les opérandes de h. Dans le cas général, tous les opérandes de h ne commutant pas avec h' doivent être évalués avant ce dernier, les autres pouvant être évalués dans n'importe quel ordre.

Concrètement, l'homomorphisme renvoyé n'est donc pas une seule composition commutative, mais un ensemble d'homomorphismes structurés par des compositions normales et commutatives. En effet, s'il n'est pas possible de faire commuter deux opérations, alors il faut conserver la composition séquentielle.

5.3.3 Fermeture transitive

Il s'agit là de la réécriture la plus significative car elle s'appuie sur le schéma général d'une relation de franchissement pour l'accessibilité qui correspond à un

point fixe d'une union d'un ensemble d'événements : $(H+Id)^\star$ avec $H = \sum_i h_i$. Nous nous appuyons pour cela sur la réécriture déjà définie pour l'union $H = F + G$:

$$\begin{aligned}(H+Id)^\star(\delta) &= (F+G+Id)^\star(\delta)\\ &= (G+Id+(F+Id)^\star)^\star(\delta)\end{aligned}$$

Le bloc $(F+Id)^\star$ est par définition localement invariant à la variable courante. Il est donc directement propagé aux nœuds successeurs, où il sera récursivement évalué en utilisant à nouveau la même définition que $(H+Id)^\star$.

À nouveau, le but de cette transformation est de favoriser le parcours en blocs de plusieurs opérations. Mais ici, cet effet est décuplé du fait que les points fixes propagés sur le SDD ne remonteront qu'une fois stabilisés. Ainsi, nous économisons des créations de nœuds intermédiaires inutiles au résultat final, et donc autant d'entrées en moins dans les caches et tables d'unicité.

Il est possible d'améliorer cette évaluation en transformant la partie G. Il est en effet possible de chaîner les opérations de G, ce qui a été reporté comme étant empiriquement plus efficace qu'une évaluation standard [Cia04]. Ce chaînage peut-être écrit comme suit pour les SDD :

$$(T+Id)^\star(\delta) = (\bigcirc_{t\in T}(t+Id))^\star(\delta)$$

Ce qui nous permet d'écrire (montré graphiquement à la figure 5.5) :

$$(H+Id)^\star(\delta) = (\bigcirc_{g\in G}(g+Id)\circ(F+Id)^\star)^\star(\delta)$$

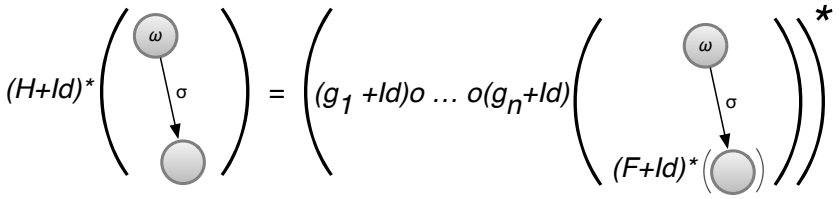


FIGURE 5.5 – Réécriture de $(H+Id)^\star$

Nous accumulons ainsi les effets du chaînage et de la saturation.

5.3.4 Applications locales

L'homomorphisme $\mathscr{L}(h,v)$ exprime le fait que h s'applique à la sous-hiérarchie contenue au niveau de la variable v. Il est donc localement invariant à toutes les autres. Nous pouvons utiliser cette information pour procéder à plusieurs transformations :

$$
\begin{aligned}
&(1) && \mathscr{L}(v,h) \circ \mathscr{L}(v,h') = \mathscr{L}(v, h \circ h') \\
&(2) && \mathscr{L}(v,h) + \mathscr{L}(v,h') = \mathscr{L}(v, h + h') \\
&(3)\ v \neq v' \Longrightarrow && \mathscr{L}(v,h) \circ \mathscr{L}(v',h') = \mathscr{L}(v',h') \circ \mathscr{L}(v,h) \\
&(4) && (\mathscr{L}(v,h) + Id)^\star = \mathscr{L}(v, (h + Id)^\star)
\end{aligned}
$$

Les expressions (1) et (2) proviennent du fait que $\mathscr{L}$ est localement invariant à toutes les variables sauf v. Ainsi, la composition ou l'union de deux applications locales à v n'aura pas d'influence autre que sur v, et donc leurs évaluations peuvent être repoussées sur la hiérarchie imbriquée. L'expression (3) exprime la commutativité de la composition de applications locales, quand elles ne s'appliquent pas à la même variable.

En effet, le seul impact de l'application $\mathscr{L}(v,h)$ est de modifier l'état de la variable v, et donc modifier v puis v' ou v' puis v a le même effet général. Donc deux applications locales ne s'appliquant pas à la même variable sont indépendantes. Nous exploitons cette possibilité pour maximiser l'effet de l'expression (1), en triant les compositions par variable touchée.

Les transformations de la fermeture transitive présentées en 5.3.3 permettent une propagation sur les successeurs d'un nœud, mais elles ne permettent pas de faire de même sur les SDD possiblement imbriqués. L'application locale $\mathscr{L}$ va nous permettre de détecter les points fixes propageables sur la hiérarchie, ce qu'exprime l'expression (4).

Nous procédons donc à une modification de la réécriture de $(H + Id)^\star(\delta)$, où $\delta = \langle \omega, \pi, \alpha \rangle$. Nous raffinons la partition d'une union H en identifiant les applications locales à ω : $H = F + L + G$ où F contient les opérations localement invariantes à ω, $L = \mathscr{L}(\omega, l)$ représente les opérations purement locales à ω, et G contient les opérations qui ne se propagent plus. En utilisant l'expression (4), nous pouvons écrire :

$$
\begin{aligned}
(H + Id)^\star(\delta) &= (F + L + G + Id)^\star(\delta) \\
&= (G + Id + (L + Id)^\star + (F + Id)^\star)^\star(\delta) \\
&= (\bigcirc_{g \in G}(g + Id) \circ \mathscr{L}(\omega, (l + Id)^\star) \circ (F + Id)^\star)^\star(\delta)
\end{aligned}
$$

Cette fois-ci, nous poussons l'évaluation des blocs propageables sur deux axes : la hiérarchie et la hauteur du SDD.

5.3.5 Fermeture transitive avec sélecteurs

Nous généralisons maintenant la saturation automatique de la fermeture transitive en prenant en compte les sélecteurs. Considérons la forme suivante, avec $H = \sum_i h_i$ et S une composition de sélécteurs :

$$(S \circ H + Id)^\star$$

Sous cette forme, la composition empêche la réécriture de la fermeture transitive. Pour palier à ce manque, nous créons une partition sur H de la façon suivante :

$$C = \{h_i \in H | commutatives(h_i, S)\}$$
$$\overline{C} = \{h_i \in H | \neg commutatives(h_i, S)\}$$

Nous avons donc $(S \oplus C + S \circ \overline{C} + Id)^\star$. S et C commutant, il est possible de les évaluer dans n'importe quel ordre. Notre objectif étant de pouvoir créer un effet de saturation, nous voulons pouvoir faire une fermeture transitive sur C. Dans ce cas, il devient primordial d'appliquer les sélecteurs avant cette fermeture transitive car il ne faut pas que C puisse s'appliquer sur des chemins qu'aurait filtré S. Nous pouvons donc écrire maintenant :

$$\left((C + Id)^\star \circ S + S \circ \overline{C} + Id\right)^\star$$

La partie $(C + Id)^\star$ est alors compatible avec la réécriture de la fermeture transitive de la section précédente.

En suivant le même principe, nous pouvons transformer la fermeture transitive $(H \circ S + Id)^\star$:

$$\begin{aligned}(H \circ S + Id)^\star &= \left(C \oplus S + \overline{C} \circ S + Id\right)^\star \\ &= \left((C + Id)^\star \circ S + \overline{C} \circ S + Id\right)^\star\end{aligned}$$

Ces réécritures sont utiles dans le cas d'utilisations de filtres sur des états générés par une relation de franchissement. C'est donc un mécanisme bien adapté à la vérification de formules de logique temporelle.

5.3.6 Factorisation

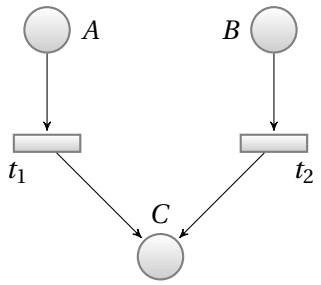


FIGURE 5.6 – Réseau de Petri R avec deux transitions partageant la même opération $h^+(C,1)$

Nous présentons dans cette section une dernière réécriture cherchant à factoriser les événements. Nous partons du principe de base du *model checking* symbolique qui génère à chaque pas de la fermeture transitive un ensemble d'états. L'idéal est de générer un maximum d'états à chaque pas afin que la génération soit la plus rapide possible.

Il n'est pas rare que des événements partagent les mêmes opérations à appliquer sur des sous-composants du système modélisé. Nous voulons donc permettre à ces opérations de s'appliquer sur un espace d'états plus grand. Considérons à titre d'exemple le réseau de Petri R de la figure 5.6. La relation de transition globale le décrivant est :

$$\begin{aligned} H_R &= \left(h^+(C,1) \circ h^-(A,1) + h^+(C,1) \circ h^-(B,1) + Id\right)^\star \\ &= \left(h^+(C,1) \circ (h^-(A,1) + h^-(B,1)) + Id\right)^\star \end{aligned}$$

En factorisant la relation de transition de cette manière, $h^+(C,1)$ va pouvoir être évalué sur un nombre d'états plus grands, puisque découverts ensemble par l'union de $h^-(A,1)$ et $h^-(B,1)$.

Cette réécriture s'effectue au sein de l'homomorphisme d'union [2], lorsqu'on détecte qu'il contient des compositions. S'agissant d'une simple factorisation, nous ne détaillons pas son fonctionnement.

Notons simplement que pour des raisons d'efficacité, nous effectuons cette transformation à l'évaluation et non pas à la construction. En effet, lors de l'évaluation de l'union, ses opérandes sont partitionnés en trois parties (*cf.* section 5.3.1) : deux se propageant (F et L), et une s'appliquant au niveau courant (G). L'opération de factorisation nécessitant de parcourir tous les opérandes, il est judicieux qu'ils soient en nombre restreint. Or seules nous intéressent à un niveau les opérations qui s'y appliquent. Donc nous effectuons cette factorisation seulement sur la partie G d'une union. Les autres parties seront factorisées au fur et à mesure de la descente de l'évaluation.

5.4 Expérimentations

Nous présentons dans cette section les résultats de notre saturation automatique. La machine de test est dotée d'un processeur Xeon à 1,83 GHz et de 4Go de mémoire.

Les quatre modèles utilisés (le diner des philosophes, la modélisation d'un protocole de communication en anneau, une modélisation du système kanban et la modélisation d'une manufacture flexible) ont été choisis car provenant de SMART, permettant ainsi de se comparer à ce dernier.

Le tableau 5.1 montre les performances de la saturation automatique et du chaînage de transitions au sein du *model checker* PNDDD. Nous indiquons pour

2. Nous envisageons de l'effectuer aussi au sein de la fermeture transitive

			Chaînage de transitions			Saturation automatique		
Taille modèle	États #	SDD #	T (s)	Mem. (Mo)	Pic #	T (s)	Mem. (Mo)	Pic #
Diner des philosophes								
100	$4{,}9\times10^{62}$	419	1.9	112	$2{,}8\times10^{5}$	0.1	3,7	3725
200	$2{,}5\times10^{125}$	819	7,9	446	$1{,}1\times10^{6}$	0,3	7,5	7425
1000	$9{,}2\times10^{626}$	4019	–	–	–	5,8	92,2	37025
4000	7×10^{2507}	16019	–	–	–	120,9	1310	$1{,}5\times10^{5}$
Anneau								
10	$8{,}3\times10^{09}$	105	1,1	48	90043	0.1	2.4	3173
50	$1{,}7\times10^{52}$	1345	–	–	–	2,3	57,4	$2{,}1\times10^{5}$
100	$2{,}6\times10^{105}$	5145	–	–	–	17,3	395	$1{,}5\times10^{6}$
150	$4{,}5\times10^{158}$	11445	–	–	–	57,6	1272	$5{,}0\times10^{6}$
Kanban								
100	$1{,}7\times10^{19}$	511	12	145	$2{,}6\times10^{5}$	0,4	5,6	13713
200	$3{,}2\times10^{22}$	1011	96	563	1×10^{6}	2,3	16,3	47413
300	$2{,}6\times10^{24}$	1511	–	–	–	7,2	33,5	$1{,}0\times10^{5}$
700	$2{,}8\times10^{28}$	3511	–	–	–	103,6	168	$5{,}1\times10^{5}$
Manufacture flexible								
50	$4{,}2\times10^{17}$	917	13	430	$5{,}3\times10^{5}$	0,3	6,7	20979
100	$2{,}7\times10^{21}$	1817	–	–	–	1,1	19,5	69454
300	$3{,}6\times10^{27}$	5417	–	–	–	17,8	151,5	$5{,}8\times10^{5}$
500	$2{,}7\times10^{30}$	9017	–	–	–	70,6	401,2	$1{,}6\times10^{6}$

TABLE 5.1 – Performances de la saturation automatique et du chaînage de transitions au sein de PNDDD

chaque méthode de génération le temps, la mémoire nécessaire au calcul, ainsi que le nombre maximal de SDD atteint lors de la génération. Les cases contenant « – » indiquent que le calcul n'a pas pu aboutir.

Les résultats sont sans appel : quelque soit le modèle, la saturation automatique apporte des gains exponentiels en temps et en mémoire. Il devient possible de générer des modèles bien plus grands que ce que n'autorisait le chaînage de transitions.

Le tableau 5.2 présente les performances de PNDDD comparées à celles de SMART. Les cases contenant « – » indiquent que le calcul ne s'est pas fini pour cause de mémoire insuffisante. Mis à part le modèle du protocole en anneau, tous les exemples nous sont favorables.

On remarque que pour les deux premier modèles, la consommation mémoire est plus importante pour PNDDD. Cela est à notre sens un problème plus technique, car une nouvelle implémentation prototype a montré qu'il était possible d'utiliser 5 à 10 fois moins de mémoire en moyenne. Par contre, les raisons des temps de calcul plus mauvais sont encore inexpliquées et restent à investiguer.

Pour les deux autres modèles, le fait que PNDDD obtienne de meilleurs résultats est expliqué par la structure même des diagrammes de décision utilisés. En

		PNDDD		SMART	
Taille modèle	États #	T (s)	Mem. (Mo)	T (s)	Mem. (Mo)
Diner des philosophes					
100	$4{,}9\times10^{62}$	0.1	3,7	0,4	3,6
1000	$9{,}2\times10^{626}$	5,8	92,2	4,6	23,6
4000	$7{,}0\times10^{2507}$	120,9	1310	–	–
Anneau					
50	$1{,}7\times10^{52}$	2,3	57,4	0,9	3,8
100	$2{,}6\times10^{105}$	17,3	395	5,1	6,7
250	16×10^{264}	–	–	155,4	24,2
Kanban					
50	$1{,}0\times10^{16}$	0,1	2,7	37,2	58,4
100	$1{,}7\times10^{19}$	0,4	11,0	–	–
Manufacture flexible					
50	$4{,}2\times10^{17}$	0,4	16	570,85	142,8
100	$2{,}7\times10^{21}$	1,9	50	–	–

TABLE 5.2 – Performances comparées de PNDDD et SMART

effet, SMART utilise des MDD qui nécessitent un arc par valeur de variable, même si elles vont vers le même successeur. Les SDD au contraire n'utilisent qu'un arc par successeur. Or les deux derniers modèles ont des domaines de variable grands, ce qui est défavorable aux MDD.

5.5 Systèmes de transition instantiables

Au cours des travaux sur la saturation automatique, la question de l'adéquation des formalismes de modélisations aux SDD s'est posée. Deux points ont mis en évidence que la notion de hiérarchie correctement exploitée permettait d'aboutir à une utilisation optimale de la saturation automatique sur les SDD :

- la formalisation de l'application locale $\mathscr{L}$ ainsi que la propagation des fermetures transitives sur les niveaux de hiérarchie d'un SDD permet de saturer des composants locaux d'un système ;
- dans [HTMK08], nous exposons un encodage récursif du modèle des philosophes, permettant de calculer l'espace d'états de 2^{30000} philosophes, en 36 secondes et 386 Mo de RAM.

Un formalisme approprié aux SDD permettrait donc d'exprimer des systèmes comme des compositions de sous-systèmes, eux-même étant potentiellment des compositions. Cela permettrait d'utiliser au mieux les capacités hiérarchiques des SDD.

Nous avons pris part à l'inception d'un tel formalisme, que Yann Thierry-Mieg a ensuite conçu. Ce formalisme, *Instantiable Transition Systems* (ITS, systèmes de transition instanciables), permet de composer hiérarchiquement des sous-systèmes. La dynamique du système est décrite en synchronisant les événements de chaque sous-système, s'inspirant des algèbres de processus. L'espace d'états global est donc un produit cartésien des espaces d'états de chaque sous-système.

[TMPHK09] introduit ce formalisme et montre son application aux modèles « réguliers », c'est à dire des modèles - tels que les philosophes - que l'on peut modéliser en composant récursivement des sous-systèmes, au moins en partie.

Les ITS sont basés sur deux notions : les types ITS et les composites ITS. Les types possèdent un espace d'états, une relation de franchissement purement locale et des transitions de synchronisation, pour intéragir avec d'autres types. Un type correspond ainsi à un sous-système. Chaque type peut-être instancié autant de fois que nécessaire. Les types ITS peuvent être composés au sein de composites ITS, qui sont eux-mêmes des types, permettant ainsi un encodage récursif.

Dans le cas de modèles réguliers, on observe que l'utilisation de ce formalisme permet de générer dans un temps et une quantité de mémoire croissant de façon polynomiale par rapport à la taille du modèle. Dans le cas des philosophes, la croissance du temps de calcul et de la quantité de mémoire se fait même de manière logarithmique.

L'usage d'un encodage hiérarchique est donc une excellente stratégie pour tirer le meilleur de la saturation automatique.

5.6 Conclusion

Dans ce chapitre, nous avons présenté une méthode permettant d'activer la technique dite de « saturation » automatiquement, en exigeant le moins d'informations possibles de l'utilisateur. Lorsque des informations lui sont demandées, elles ne sont en fait que la formalisation de la logique qu'il a conçue. Par exemple, la localité d'un homomorphisme demandée au travers du prédicat V est une information qu'il maîtrise nécessairement.

Pour activer la saturation automatique, nous avons identifié des catégories remarquables d'homomorphismes, qui en combinaison avec des transformations ont permis essentiellement de :

– créer des « paquets » d'opérations traversant d'un seul bloc les SDD ;
– créer des fermetures transitives ciblées, en hiérarchie ou non.

Les homomorphismes remarquables, que nous avons introduits permettent à l'utilisateur de mieux structurer ses opérations. Par exemple, l'application locale $\mathscr{L}$ lui permet d'exprimer simplement une application en hiérarchie. Ou encore, l'homomorphismes ITE lui permet d'appliquer de manière conditionnelle des opérations.

Nous avons généralisé le principe de la saturation en étant capable de l'appliquer non plus sur la seule fermeture transitive extérieure de génération d'un es-

pace d'états, mais sur toute fermeture transitive, imbriquée ou non. Cela nous permet d'appliquer la saturation lors de la vérification de formules de logiques temporelles, qui sont composées du plusieurs fermetures transitives. De plus, nous ne dépendons d'aucune analyse préalable des modèles vérifiés, ni d'aucun formalisme.

Enfin, nous présentons des performances exponentiellement meilleures par rapport à la méthode plus classique de chaînage de transitions. Aussi, nous sommes comparables au *model checker* SMART qui est à l'origine de la saturation.

CHAPITRE 6

Conclusion générale

L'**explosion combinatoire** est le problème principal du *model checking*. Ces travaux apportent deux types de solutions à ce problème :

- l'exploitation du parallélisme pour le *model checking* explicite ;
- un mécanisme activant automatiquement des techniques de franchissement optimisées, pour le *model checking* symbolique.

Nos solutions ont l'avantage d'être transparentes à l'utilisateur.

6.1 Contributions

Parallélisme pour le *model checking* explicite. L'objectif est d'utiliser les ressources combinées de plusieurs machines pour obtenir à la fois une puissance de calcul accrue et une plus grande quantité de mémoire.

À partir des critères de conception d'un *model checker* réparti issus de la littérature, nous proposons une architecture dédiée à la vérification de propriétés d'accessibilité. Nous exploitons cette architecture pour répartir la génération d'espaces d'états avec GreatSPN.

Ces expérimentations avec GreatSPN sont couronnées de succès : les gains obtenus sont quasi-linéaires (voire supra-linéaires sur des architectures spécifiques). Ces résultats requièrent des critères spécifiques que nous avons mis en évidence. Ils permettent d'obtenir un **recouvrement** des communications par les temps de calculs, permettant ainsi d'obtenir une **accélération linéaire**.

Ces critères sont simples : les règles de franchissement doivent avoir un coût de calcul élevé et il faut utiliser simultanément plusieurs *threads* de calcul. Nous pensons que le nombre de messages échangés sur le réseau est loin d'être aussi primordial que le laisse entendre la littérature.

Franchissements optimisés pour le *model checking* symbolique. Nous utilisons les **SDD** manipulés à l'aide d'homomorphismes. L'objectif est d'utiliser la

technique dite de « **saturation** », introduite par [CLS01b], qui permet des gains en mémoire et en temps potentiellement exponentiels.

Nous avons généralisé cette technique en lui permettant de s'appliquer à toute fermeture transitive d'une somme d'événements, en conjonction avec des filtres. De plus, nous pouvons l'appliquer sur des imbrications de fermetures transitives. Cela nous permet de sortir du cadre de l'accessibilité. Cette technique devient utilisable pour la vérification de formules de logique temporelle, ou toute autre application faisant usage de fermetures transitives.

L'usage des homomorphismes n'est pas restreint aux seuls SDD : ce travail est donc généralisable à d'autres diagrammes de décision.

Transparence à l'utilisateur. Toutes nos contributions sont transparentes à l'utilisateur :

- pour répartir un *model checker* il suffit de fournir la relation de franchissement, en respectant quelques contraintes d'ordre techniques ;
- l'activation de la saturation est également automatique à partir du moment où l'utilisateur fournit les informations permettant de détecter des motifs exploitables pour la réécriture.

Pour ce qui est de la saturation automatique, l'utilisateur maîtrise complètement les informations que nous lui demandons car elles ne sont que la formalisation de concepts qu'il manipule déjà. La détection des motifs activant la saturation s'appuie sur de nouveaux homomorphismes, issus de notre expérience. Ils correspondent à des « bonnes pratiques » d'écritures d'homomorphismes permettant à l'utilisateur d'exprimer de manière concise des opérations qui sont « utilitaires » pour se concentrer sur ses opérations de manipulation.

6.2 Perspectives

Ces résultats ouvrent plusieurs perspectives :

***Model checking* adaptatif.** De prime abord, l'apport du *model checking* parallèle et réparti semble limité en regard des gains exponentiels de la saturation automatique. Cependant, la première méthode s'applique au *model checking* explicite, alors que la deuxième est purement symbolique. Il est de plus notoire que les techniques symboliques sont plus difficiles à maîtriser que celles explicites. Enfin, ces techniques ne s'appliquent pas avec autant de succès à chaque type de modèle.

Ainsi, outre l'élaboration de méthodes efficaces de *model checking*, il faut s'intéresser aux moyens d'identifier les conditions qui les rendent exploitables. Dans ce contexte, la notion de *model checking* **adaptatif** est une piste naturelle : les modèles sont pré-analysés afin d'identifier la combinaison de techniques adaptées, un *model checker* particularisé est alors construit pour traiter le modèle. Par exemple, le domaine des variables d'un SDD n'est pas imposé : on peut référencer des BDD ou une représentation d'états explicites sur les arcs d'un SDD. Une

telle stratégie peut s'avérer intéressante dans le cas de systèmes GALS (*Globally Asynchronous, Locally Synchronous*) ou de « systèmes de systèmes ».

Ordonnancement des variables. L'ordre des variables est primordial pour l'efficacité des techniques à base de diagrammes de décisions. Nous avons utilisé les informations de structuration fournies par les homomorphismes pour automatiser la saturation. Nous pensons qu'il est également possible d'exploiter cette structuration afin de déterminer un bon ordre.

La hiérarchie des SDD rajoute un degré de complexité à cet ordre de variables. Il faut en effet aussi déterminer quel découpage est le plus adapté pour un modèle donné. Les ITS, grâce à leur modélisation par composition, permettent déjà d'établir un découpage en fonction du modèle. Nous estimons qu'il est possible d'utiliser la structure des homomorphismes s'appliquant au modèle pour affiner la gestion de cette hiérarchie (*i.e.* agréger ou découper des composantes).

Parallélisation et répartition des SDD. Les deux techniques que nous présentons dans ce mémoire sont difficilement conjugables. La littérature est émaillée de tentatives de répartitions de *model checkers* symboliques, sans jamais obtenir de grand succès. L'auteur initial de la saturation a également essayé de la paralléliser [CC04, CC05, CC06], mais avec des résultats mitigés. La raison principale de ces échecs est le caractère compact des diagrammes de décision qui implique de fortes synchronisations entre les *threads* de calcul.

Nous pensons que la hiérarchie des SDD est à même de réussir là où les autres ont échoué. En effet, si chaque niveau de hiérarchie possède son propre espace mémoire, il est alors possible d'y faire travailler un *thread* qui se synchroniserait plus rarement avec les autres.

Autres fermetures transitives. Nous avons présenté plusieurs types de fermetures transitives que nous savons rendre optimales pour appliquer la saturation. Elles sont le fruit d'une expérience d'utilisation. Mais il est certain que d'autres formes de fermetures se présenteront à l'avenir. Il faudra donc d'optimiser chacune d'entre elles au fur et à mesure de leurs apparitions.

Bibliographie

[ADK98] Allmaier, S. C., S. Dalibor et D. Kreische: *Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines*. Dans D'Hollander, E. H., G. R. Joubert, F. J. Peters et U. Trottenberg (rédacteurs) : *Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, tome 12, pages 581–588, Amsterdam, 1998. Elsevier. citeseer.ist.psu.edu/article/allmaier97parallel.html.

[AKH97] Allmaier, S., M. Kowarschik et G. Horton: *State space construction and steady-state solution of GSPNs on a shared–memory multiprocessor*, 1997. citeseer.ist.psu.edu/allmaier97state.html.

[AS89] Alpem, Bowen et Fred B. Schneider: *Verifying temporal properties without temporal logic*. ACM Transactions on Programming Languages, 11 :147–167, 1989.

[BBCŠ05] Barnat, J., Luboš Brim, Ivana Cerná et P. Šimeček: *DiVinE The Distributed Verification Environment*. Dans Leucker, M. et J. van de Pol (rédacteurs) : *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portuga, 2005.

[BCL91] Burch, Jerry R., Edmund M. Clarke et David E. Long: *Symbolic Model Checking with Partitioned Transistion Relations*. Dans *VLSI*, pages 49–58, 1991.

[BCM+90] Burch, Jerry R., Edmund M. Clarke, Kenneth L. McMillan, David L. Dill et L. J. Hwang: *Symbolic Model Checking :* 10^{20} *States and Beyond*. Dans *LICS*, pages 428–439. IEEE Computer Society, 1990.

[Beh02] Behrmann, Gerd: *A Performance Study of Distributed Timed Automata Reachability Analysis*. Electronic Notes in Theoretical Computer Science, 68 :486–502, 2002, ISSN 4. http://www.sciencedirect.com/science/article/B75H1-4G1TW61-N/2/cb7f084efad64a173687cb7bec89202b.

[BFLW05] Barnat, J., V. Forejt, M. Leucker et Michael Weber: *DivSPIN - A SPIN compatible distributed model checker.* Dans Leucker, M. et J. van de Pol (rédacteurs) : *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portuga, 2005.

[BHSV+96] Brayton, Robert, Gary Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev Ranjan, Shaker Sarwary, Thomas Staple, Gitanjali Swamy et Tiziano Villa: *VIS : A system for verification and synthesis.* Computer Aided Verification, pages 428–432, 1996. http://dx.doi.org/10.1007/3-540-61474-5_95.

[BLW01] Bollig, Benedikt, Martin Leucker et Michael Weber: *Parallel Model Checking for the Alternation Free μ-Calculus.* Dans *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, tome 2031 de *Lecture Notes in Computer Science*, pages 543–558, 2001. citeseer.nj.nec.com/421114.html.

[Bry86] Bryant, Randal E.: *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, C-35(8) :677–691, 1986. citeseer.ist.psu.edu/article/bryant86graphbased.html.

[Bry92] Bryant, Randal E.: *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.* ACM Computing Surveys, 24(3) :293–318, 1992. citeseer.ist.psu.edu/bryant92symbolic.html.

[BW96] Bollig, Beate et Ingo Wegener: *Improving the Variable Ordering of OBDDs Is NP-Complete.* IEEE Trans. Comput., 45(9) :993–1002, 1996, ISSN 0018-9340.

[CC04] Chung, Ming Ying et Gianfranco Ciardo: *Saturation NOW.* Dans *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, ISBN 0-7695-2185-1.

[CC05] Chung, Ming Ying et Gianfranco Ciardo: *A Pattern Recognition Approach for Speculative Firing Prediction in Distributed Saturation State-Space Generation.* Electronic Notes in Theoretical Computer Science, 135(2) :65–80, 2005. http://www.sciencedirect.com/science/article/B75H1-4J77J57-6/2/eb29e645df76a25ae110f10f01556319.

[CC06] Chung, Ming Ying et G. Ciardo: *A dynamic firing speculation to speedup distributed symbolic state-space generation.* pages 10 pp.–, 2006.

[CCBF94] Caselli, Stefano, G. Conte, F. Bonardi et M. Fontanesi: *Experiences on SIMD massively parallel GSPN analysis*, tome 794. 1994. http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/3-540-58021-2_15.

[CCD+05] Cappello, Franck, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Yvon Jégou, Pascale Vicat Blanc Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Benjamin Quétier et Olivier Richard: *Grid'5000 : a large scale and highly reconfigurable grid experimental testbed*. Dans *GRID*, pages 99–106, 2005.

[CCGR99] Cimatti, A., E. Clarke, F. Giunchiglia et M. Roveri: *NuSMV : A New Symbolic Model Verifier*. Computer Aided Verification, pages 682–682, 1999. http://dx.doi.org/10.1007/3-540-48683-6_44.

[CCM95] Caselli, Stefano, Gianni Conte et P. Marenzoni: *Parallel state space exploration for GSPN models*, tome 935. 1995. http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/3-540-60029-9_40.

[CCM01] Caselli, Stefano, G. Conte et P. Marenzoni: *A Distributed Algorithm for GSPN Reachability Graph Generation*. Journal of Parallel and Distributed Computing, 61(1) :79–95, 2001. http://www.sciencedirect.com/science/article/B6WKJ-457CHYJ-43/2/b7806be12e9e7229f6e57551178d8532.

[CDFH91] Chiola, G., C. Dutheillet, G. Franceschini et S. Haddad: *On Well-Formed Coloured Nets and their Symbolic Reachability Graph*. High-Level Petri Nets. Theory and Application, LNCS, 1991.

[CDFH93] Chiola, Giovanni, Claude Dutheillet, Giuliana Franceschinis et Serge Haddad: *Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications*. IEEE Trans. Computers, 42(11) :1343–1360, 1993.

[CEJS98a] Clarke, Edmund M., E. Allen Emerson, Somesh Jha et A. Prasad Sistla: *Symmetry Reductions in Model Checking*. Dans *CAV*, pages 147–158, 1998.

[CEJS98b] Clarke, Edmund M., E. Allen Emerson, Somesh Jha et A. Prasad Sistla: *Symmetry Reductions inModel Checking*. Dans Hu, Alan J. et Moshe Y. Vardi (rédacteurs) : *CAV*, tome 1427 de *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998, ISBN 3-540-64608-6.

[CEPA+02] Couvreur, Jean Michel, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud et Pierre André Wacrenier: *Data Decision Diagrams for Petri Net Analysis*. Application and Theory of Petri

Nets 2002, pages 129–158, 2002. http://dx.doi.org/10.1007/3-540-48068-4_8.

[CFGR95] Chiola, Giovanni, Giuliana Franceschinis, Rossano Gaeta et Marina Ribaudo: *GreatSPN 1.7 : Graphical Editor and Analyzer for Timed and Stochastic Petri Nets*. Perform. Eval., 24(1-2) :47–68, 1995.

[CGJ+00] Clarke, Edmund M., Orna Grumberg, Somesh Jha, Yuan Lu et Helmut Veith: *Counterexample-Guided Abstraction Refinement*. Dans *CAV*, pages 154–169, 2000.

[CGL94] Clarke, Edmund M., Orna Grumberg et David E. Long: *Model checking and abstraction*. ACM Trans. Program. Lang. Syst., 16(5) :1512–1542, 1994, ISSN 0164-0925.

[CGN98] Ciardo, Gianfranco, Joshua Gluckman et David Nicol: *Distributed state-space generation of discrete-state stochastic models*. INFORMS J. Comp., 10(1) :82–93, 1998.

[Cia01] Ciardo, Gianfranco: *Distributed and Structured Analysis Approaches to Study Large and Complex Systems*. Lecture Notes in Computer Science, 2090 :344–??, 2001. citeseer.ist.psu.edu/ciardo01distributed.html.

[Cia04] Ciardo, Gianfranco: *Reachability Set Generation for Petri Nets : Can Brute Force Be Smart ?* Applications and Theory of Petri Nets 2004, pages 17–34, 2004. http://www.springerlink.com/content/8f0cv94wrf8cnhny.

[CLM07] Ciardo, Gianfranco, Gerald Lüttgen et Andrew Miner: *Exploiting interleaving semantics in symbolic state-space generation*. Formal Methods in System Design, 31(1) :63–100, 2007. http://dx.doi.org/10.1007/s10703-006-0033-y.

[CLS01a] Ciardo, Gianfranco, Gerald Lüttgen et Radu Siminiceanu: *Saturation : An Efficient Iteration Strategy for Symbolic State-Space Generation*. Tools and Algorithms for the Construction and Analysis of Systems : 7th International Conference, TACAS 2001 : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings :, pages 328–, 2001. http://www.springerlink.com/content/mbff40ngvw3m8k2b.

[CLS01b] Ciardo, Gianfranco, Gerald Lüttgen et Radu Siminiceanu: *Saturation : An Efficient Iteration Strategy for Symbolic State-Space Generation*. Tools and Algorithms for the Construction and Analysis of Systems : 7th International Conference, TACAS 2001 : Held

as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings :, pages 328–, 2001. http://www.springerlink.com/content/mbff40ngvw3m8k2b.

[CLY07] Ciardo, Gianfranco, Gerald Lüttgen et Andy Jinqing Yu: *Improving Static Variable Orders via Invariants*. Dans Grumberg, Orna et Michael Huth (rédacteurs) : *Proc. of the 13th International Conference, TACAS 2007*, numéro 4424 dans *LNCS*. Springer, 2007.

[CM97] Ciardo, Gianfranco et Andrew S. Miner: *Storage Alternatives for Large Structured State Spaces*. Rapport technique TR-97-5, 1997. citeseer.ist.psu.edu/ciardo97storage.html.

[CM04] Ciardo, Gianfranco et Andrew S. Miner: *SMART : The Stochastic Model checking Analyzer for Reliability and Timing*. Dans *QEST*, pages 338–339. IEEE Computer Society, 2004, ISBN 0-7695-2185-1.

[CMB90] Coudert, Olivier, Jean Christophe Madre et Christian Berthet: *Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams*. Dans Clarke, Edmund M. et Robert P. Kurshan (rédacteurs) : *CAV*, tome 531 de *Lecture Notes in Computer Science*, pages 23–32. Springer, 1990.

[CMS03] Ciardo, Gianfranco, Robert Marmorstein et Radu Siminiceanu: *Saturation Unbound*. Tools and Algorithms for the Construction and Analysis of Systems, pages 379–393, 2003. http://dx.doi.org/10.1007/3-540-36577-X_27.

[Coq] *http ://coq.inria.fr*. http://coq.inria.fr.

[CTM05] Couvreur, Jean Michel et Yann Thierry-Mieg: *Hierarchical Decision Diagrams to Exploit Model Structure*. Formal Techniques for Networked and Distributed Systems - FORTE 2005, pages 443–457, 2005. http://dx.doi.org/10.1007/11562436_32.

[DFvG83] Dijkstra, Edsger W., W. H. J. Feijen et A. J. M. van Gasteren: *Derivation of a Termination Detection Algorithm for Distributed Computations*. Inf. Process. Lett., 16(5) :217–219, 1983.

[DLP04] Duret-Lutz, Alexandre et Denis Poitrenaud: *SPOT : an Extensible Model Checking Library using Transition-based Generalized Büchi Automata*. Dans *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83, Volendam, The Netherlands, octobre 2004. IEEE Computer Society Press.

[ELS07] Ezekiel, J., G. Lüttgen et R. Siminiceanu: *Can Saturation be Parallelised ? On the Parallelisation of a Symbolic State-Space Generator*. Dans Brim, L., B. Haverkort, M. Leucker et J. van de Pol (rédacteurs) : *5th Intl. Workshop on Parallel and Distributed Methods of Verification (PDMC 2006)*, tome 4346 de *Lecture Notes in Computer Science*, pages 331–346, Bonn, Germany, August 2007. Springer-Verlag.

[Est] Esterel Technologies: *http ://www.esterel-technologies.com/products/scade-suite*. http://www.esterel-technologies.com/products/scade-suite/.

[Fly72] Flynn, M. J.: *Some Computer Organizations and Their Effectiveness*. IEEE Trans. Computers, C-21(9) :948–960, septembre 1972.

[Gar98] Garavel, H.: *OPEN/CAESAR : An Open Software Architecture for Verification, Simulation, and Testing*. Dans Steffen, Bernhard (rédacteur) : *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98,*, 1998. citeseer.ist.psu.edu/garavel98opencaeligsar.html.

[GBD+94] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek et Vaidy Sunderam: *PVM*. MIT Press, Cambridge, MA., 1994.

[GHIS05] Grumberg, Orna, Tamir Heyman, Nili Ifergan et Assaf Schuster: *Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation*. Correct Hardware Design and Verification Methods, pages 129–145, 2005. http://dx.doi.org/10.1007/11560548_12.

[GLS99] Gropp, William, Ewing Lusk et Anthony Skjellum: *Using MPI*. MIT Press, Cambridge, MA., 1999.

[GMS01] Garavel, Hubert, Radu Mateescu et Irina Smarandache: *Parallel State Space Construction for Model-Checking*, tome 2057. 2001. http://www.springerlink.com/openurl.asp?genre=article&id=63D7C9EN4BJKAYQW.

[God95] Godefroid, Patrice: *Partial Order Methods for the Verification of Concurrent Systems*. Thèse de doctorat, Université de Liège, 1995.

[GV01] Geldenhuys, Jaco et Antti Valmari: *Techniques for Smaller Intermediary BDDs*. Dans Larsen, Kim Guldstrand et Mogens Nielsen (rédacteurs) : *CONCUR*, tome 2154 de *Lecture Notes in Computer Science*, pages 233–247. Springer, 2001, ISBN 3-540-42497-0.

[GV03] Girault, Claude et Rüdiger Valk: *Petri Nets for Systems Engineering*. Springer, 2003.

[HB07] Holzmann, Gerard J. et Dragan Bosnacki: *Multi-Core Model Checking with SPIN*. pages 1–8, 2007.

[HJJJ85] Huber, Peter, Arne M. Jensen, Leif O. Jepsen et Kurt Jensen: *Towards reachability trees for high-level Petri nets*. Dans *LNCS 188*, pages 215–233. Springer-Verlag, 1985.

[HKTMLA07] Hamez, Alexandre, Fabrice Kordon, Yann Thierry-Mieg et Fabrice Legond-Aubry: *dmcG : A Distributed Symbolic Model Checker Based on GreatSPN*. Petri Nets and Other Models of Concurrency –ICATPN 2007, pages 495–504, 2007. http://dx.doi.org/10.1007/978-3-540-73094-1_29.

[HN96] Harel, D. et A. Naamad: *The STATEMATE Semantics of Statecharts*. ACM Trans. Softw. Eng. Methodol., 5(4) :293–333, 1996.

[Hol97] Holzmann, Gerard J.: *The Model Checker SPIN*. Software Engineering, 23(5) :279–295, 1997. citeseer.ist.psu.edu/holzmann97model.html.

[HST09] Heiner, Monika, Martin Schwarick et Alexej Tovchigrechko: *DSSZ-MC - A Tool for Symbolic Analysis of Extended Petri Nets*. Dans Franceschinis, Giuliana et Karsten Wolf (rédacteurs) : *Petri Nets*, tome 5606 de *Lecture Notes in Computer Science*, pages 323–332. Springer, 2009, ISBN 978-3-642-02423-8.

[HTMK08] Hamez, Alexandre, Yann Thierry-Mieg et Fabrice Kordon: *Hierarchical Set Decision Diagrams and Automatic Saturation*. Applications and Theory of Petri Nets, pages 211–230, 2008. http://dx.doi.org/10.1007/978-3-540-68746-7_16.

[HTMK09] Hamez, A., Y. Thierry-Mieg et F. Kordon: *Building Efficient Model Checkers using Hierarchical Set Decision Diagrams and Automatic Saturation*. Fundamenta Informaticae, 94(3-4) :413–437, September 2009.

[HVP+05] Hugues, Jérôme, Thomas Vergnaud, Laurent Pautet, Yann Thierry-Mieg, Souheib Baarir et Fabrice Kordon: *On the Formal Verification of Middleware Behavioral Properties*. Electr. Notes Theor. Comput. Sci., 133 :139–157, 2005.

[ID93] Ip, C. Norris et David L. Dill: *Efficient Verification of Symmetric Concurrent Systems*. Dans *ICCD*, pages 230–234, 1993.

[JMB+03] Jones, Mike, Eric Mercer, Tonglaga Bao, Rahul Kumar et Peter Lamborn: *Benchmarking Explicit State Parallel Model Checkers*. Electr. Notes Theor. Comput. Sci., 89(1), 2003.

[Jou03] Joubert, Christophe: *Distributed Model Checking : From Abstract Algorithms to Concrete Implementations*. Electronic Notes in Theoretical Computer Science, 89(1) :114–127, 2003. http://www.sciencedirect.com/science/article/B75H1-4G7MXP0-B/2/6737ce8a12f612976b6b685a62b879a9.

[KJM04] Kumar, R., M. D. Jones et E. G. Mercer: *Dynamic Partition Algorithm for Distributed Murphi*. Rapport technique VV-0402 (also VV-03), Dept. of Computer Science, Brigham Young U., 2004.

[KM03] Koufaty, David et Deborah T. Marr: *Hyperthreading Technology in the Netburst Microarchitecture*. IEEE Micro, 23(2) :56–65, 2003, ISSN 0272-1732.

[KM05] Kumar, Rahul et Eric G. Mercer: *Load Balancing Parallel Explicit State Model Checking*. Electr. Notes Theor. Comput. Sci., 128(3) :19–34, 2005.

[KP04] Kristensen, Lars M. et Laure Petrucci: *An Approach to Distributed State Space Exploration for Coloured Petri Nets*. Dans *Proceedings of Applications and Theory of Petri Nets 2004 : 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004 — Volume 3099 of Lecture Notes in Computer Science / Cortadella, Reisig (Eds.)*, pages 474–483. Springer-Verlag, 2004.

[KP08] Klai, Kais et Denis Poitrenaud: *MC-SOG : An LTL Model Checker Based on Symbolic Observation Graphs*. Dans *Petri Nets*, pages 288–306, 2008.

[Kum04] Kumar, R.: *Load Balancing Parallel Explicit State Model Checking*. Thèse de doctorat, Brigham Young University, 2004.

[Ler00] Lerda, Flavio: *Model checking : Tecniche di verifica formalae in ambiente distributo*. Mémoire de maîtrise, Politecnico di Torino, 2000.

[Lin09] Linard, Alban: *Sémantique paramétrable des Diagrammes de Décision : une démarche vers l'unification*. Thèse de doctorat, Laboratoire d'Informatique de Paris 6, 2009.

[LS99] Lerda, Flavio et Riccardo Sisto: *Distributed-memory Model Checking with SPIN*. Dans *Proc. of the 5th International SPIN Workshop*, tome 1680 de *LNCS*. Springer-Verlag, 1999.

[LV01] Lerda, Flavio et Willem Visser: *Addressing Dynamic Issues of Program Model Checking*. Lecture Notes in Computer Science, 2057, 2001. citeseer.ist.psu.edu/article/lerda01addressing.html.

[Mat87] Mattern, F.: *Algorithms for distributed termination detection*. Distributed Computing, 2(3) :161–175, 1987.

[MC99] Miner, Andrew S. et Gianfranco Ciardo: *Efficient Reachability Set Generation and Storage Using Decision Diagrams*. Applications and Theory of Petri Nets 1999 : 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA, June 1999. Proceedings, pages 691–691, 1999. http://www.springerlink.com/content/m9f5n8017dx8xule.

[MCC97] Marenzoni, P., Stefano Caselli et G. Conte: *Analysis of large GSPN models : a distributed solution tool*. Dans *PNPM '97 : Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 122, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-7931-X. http://doi.ieeecomputersociety.org/10.1109/PNPM.1997.595543.

[MPS+06] Melatti, Igor, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby et Ganesh Gopalakrishnan: *Parallel and Distributed Model Checking in Eddy*. Dans *SPIN*, pages 108–125, 2006.

[NC97] Nicol, David M. et Gianfranco Ciardo: *Automated Parallelization of Discrete State-Space Generation*. Journal of Parallel and Distributed Computing, 47(2) :153–167, 1997. citeseer.ist.psu.edu/nicol02automated.html.

[Nic95] Nicol, David M.: *Noncommittal barrier synchronization*. Parallel Computing, 21(4) :529–549, 1995. citeseer.ist.psu.edu/nicol95noncommittal.html.

[Ope] Open Group: *POSIX Threads*. http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html.

[ORS92] Owre, S., J. M. Rushby, et N. Shankar: *PVS : A Prototype Verification System*. Dans Kapur, Deepak (rédacteur) : *11th International Conference on Automated Deduction (CADE)*, tome 607 de *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. http://www.csl.sri.com/papers/cade92-pvs/.

[Pel04] Pelánek, Radek: *Typical Structural Properties of State Spaces*. Dans *Proc. of SPIN Workshop*, tome 2989 de *LNCS*, pages 5–22. Springer-Verlag, 2004.

[Pla85] Plateau, Brigitte: *On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms*. Dans *SIGMETRICS*, pages 147–154, 1985.

[Poi96] Poitrenaud, Denis: *Graphe de Processus Arborescents pour la Vérification de Propriétés*. Thèse de doctorat, Université Pierre et Marie Curie, décembre 1996.

[PPP07] Pajault, Christophe et Jean François Pradat-Peyre: *Distributed Colored Petri Net Model-Checking with Cyclades*. Formal Methods : Applications and Technology, pages 347–361, 2007. http://dx.doi.org/10.1007/978-3-540-70952-7_24.

[PRCB94] Pastor, Enric, Oriol Roig, Jordi Cortadella et Rosa Badia: *Petri net analysis using boolean manipulation*. Application and Theory of Petri Nets 1994, pages 416–435, 1994. http://dx.doi.org/10.1007/3-540-58152-9_23.

[RCP95] Roig, Oriol, Jordi Cortadella et Enric Pastor: *Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets*. Dans Michelis, Giorgio De et Michel Diaz (rédacteurs) : *Application and Theory of Petri Nets*, tome 935 de *Lecture Notes in Computer Science*, pages 374–391. Springer, 1995, ISBN 3-540-60029-9.

[RDBSC04] Rangarajan, Murali, Samar Dajani-Brown, Kirk Schloegel et Darren Cofer: *Analysis of Distributed Spin Applied to Industrial-Scale Models*. Model Checking Software, pages 267–285, 2004. http://www.springerlink.com/content/6n608ha49rd4t02q.

[Sch03] Schmidt, Karsten: *Distributed Verification with LoLA*. Fundam. Inform., 54(2-3) :253–262, 2003. http://iospress.metapress.com/(hy5rhwjbuj1awuidvozhxveb)/app/home/contribution.asp?referrer=parent&backto=searcharticlesresults,1,1;.

[SD97] Stern, Ulrich et David L. Dill: *Parallelizing the Murφ Verifier*. Dans *Computer Aided Verification*, pages 256–278, 1997. http://citeseer.ist.psu.edu/stern97parallelizing.html.

[SHMB90] Srinivasan, A., T. Ham, S. Malik et R.K. Brayton: *Algorithms for discrete function manipulation*. Dans *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95, Nov 1990.

[ST98] Strehl, K. et L. Thiele: *Symbolic model checking using interval diagram techniques*. rapport technique, Computer Engineering and Networks Lab (TIK) Swiss Federal Institute of Technology (ETH) Zurich, 1998.

[TM04] Thierry-Mieg, Y.: *Techniques pour le Model-Checking de spécifications de Haut Niveau*. Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, 2004.

[TMPHK09] Thierry-Mieg, Yann, Denis Poitrenaud, Alexandre Hamez et Fabrice Kordon: *Hierarchical Set Decision Diagrams and Regular Models*. Dans *TACAS*, pages 1–15, 2009.

[Tov08] Tovchigrechko, Alexey: *Efficient Symbolic Analysis of Bounded Petri Nets Using Interval Decision Diagrams*. Thèse de doctorat, Der Fakultät für Mathematik, Naturwissenschaften und Informatik der Brandenburgischen Technischen Universität Cottbus, 2008.

[Val90] Valmari, Antti: *A Stubborn Attack On State Explosion*. Dans Clarke, Edmund M. et Robert P. Kurshan (rédacteurs) : *CAV*, tome 531 de *Lecture Notes in Computer Science*, pages 156–165. Springer, 1990.

[Val96] Valmari, Antti: *The State Explosion Problem*. Dans Reisig, Wolfgang et Grzegorz Rozenberg (rédacteurs) : *Petri Nets*, tome 1491 de *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996, ISBN 3-540-65306-6.

[VAM96] Vernadat, François, Pierre Azéma et François Michel: *Covering Step Graph*. Dans *Application and Theory of Petri Nets*, pages 516–535, 1996.

[WLR93] Willebeek-LeMair, M.H. et A.P. Reeves: *Strategies for Dynamic Load Balancing on Highly Parallel Computers*. IEEE Transactions on Parallel and Distributed Systems, 04(9) :979–993, 1993, ISSN 1045-9219.

[XL94] Xu, C. et F. Lau: *Iterative dynamic load balancing in multicomputers*, 1994. citeseer.ist.psu.edu/xu94iterative.html.

ANNEXE A

Interfaces de la bibliothèque
LIBDMC

L'utilisateur doit fournir trois composants :

1. Un itérateur réalisant les interfaces de `abstract_state_succ_iterator` (figure A.1), qui doit calculer les successeurs immédiats d'un état.

```
class abstract_state_succ_iterator
{
public:

        virtual ~abstract_state_succ_iterator(){};
        virtual bool has_successors() = 0;
        virtual dmc_state* get_next_successor() = 0;
};
```

FIGURE A.1 – Interfaces de `abstract_state_succ_iterator`

2. Un modèle réalisant les interfaces de `abstract_model` [1] (figure A.2). Ce modèle doit pouvoir fournir l'état initial, ainsi qu'un itérateur à partir d'un état passé en paramètre.
3. Une fabrique de modèles réalisant les interfaces de `abstract_model_factory` (figure A.3), qui correspond au design pattern «factory». Elle sera utilisée par chaque thread de génération pour instancier un objet se conformant à l'interface `abstract_model`.

`dmc_state` est une simple structure de données contenant un pointeur sur un état, ainsi que sa taille.

1. Le nom sera modifié à l'avenir, puisqu'il faudrait parler de formalisme.

```
class abstract_model
{
public:

        virtual ~abstract_model(){};
        virtual dmc_state* get_initial_state() = 0;
        virtual abstract_state_succ_iterator*
           get_succ_iterator(dmc_state* s) = 0 ;
};
```

FIGURE A.2 – Interfaces de `abstract_model`

```
class abstract_model_factory
{
public:

        virtual ~abstract_model_factory(){};
        virtual abstract_model* create_model() = 0;
};
```

FIGURE A.3 – Interfaces de `abstract_model_factory`

Résumé

Garantir la fiabilité des systèmes informatiques exige des moyens de vérification rigoureux. Le *model checking* est une technique de vérification dont l'intérêt majeur est l'automatisation, et donc la facilité d'utilisation pour les ingénieurs. La récente attribution du prix Turing aux créateurs de cette technique atteste de sa viabilité.

Le *model checking* explore exhaustivement les modèles analysés. Cela amène un problème majeur : **l'explosion combinatoire** liée aux espaces d'états des grands systèmes. Depuis plus de vingt ans, de nombreuses solutions ont été proposées pour repousser cette limite de taille afin d'être capable de traiter des espaces d'états toujours plus grands dont les tailles peuvent atteindre très rapidement les 10^{400} éléments.

Les travaux présentés ici proposent deux types de solutions pour traiter plus efficacement de plus grands espaces d'états. La première s'appuie sur les ressources de **calcul parallèle** des machines multi-processeurs, omniprésentes aujourd'hui, et des grappes de calcul. La deuxième propose de traiter plus efficacement les diagrammes de décision en automatisant la technique dite de **saturation**, dont l'efficacité empiriquement montrée est très difficile à atteindre manuellement.

Abstract

Ensuring the reliability of computer systems requires the usage of rigorous verification. Model checking is a technique of verification whose major advantage is automation, and therefore the ease of use for engineers. The recent attribution of the Turing Award to the creators of this technique proves its viability.

The model checking exhaustively explores the models analyzed. This causes a major problem : **combinatorial explosion** due to the state spaces of large systems. For over twenty years, many solutions have been proposed to push the limit of size to be able to handle spaces states increasingly large sizes which can reach very quickly the 10^{400} elements.

The work presented here offer two types of solutions to address more effectively larger spaces of states. The first relies on resources **parallel computing** machines with multiple CPUs, ubiquitous today, and cluster computing. The second proposed deal more effectively with decision diagrams by automating technique called **saturation**, whose effectiveness empirically shown is very difficult to achieve manually.