Model checking parallèle et réparti de réseaux de Petri colorés de haut-niveau : application à la vérification automatique de programmes Ada concurrents

Christophe Pajault

Thèse de doctorat de l'Université Pierre et Marie Curie

Spécialité
Informatique

Présentée par
Christophe Pajault

Pour obtenir le grade de
Docteur de l'Université Pierre et Marie Curie

Model checking parallèle et réparti de réseaux de Petri colorés de haut-niveau.

Application à la vérification automatique de programmes Ada concurrents.

Soutenue le 23 juin 2008 devant un jury composé de:

Rapporteurs
M. **Didier Buchs**, Professeur de l'Université de Genève
M. **Gaétan Hains**, Professeur de l'Université Paris 12

Examinateurs
M. **Franco Gasperoni**, Docteur de l'Université de New-York et Directeur Général d'ACT Europe
M. **Claude Girault**, Professeur émerite à l'Université Pierre et Marie Curie
M. **Denis Poitrenaud**, Maître de Conférence de l'Université Paris 5

Directeurs de thèse
M. **Claude Kaiser**, Professeur émerite au Conservatoire National des Arts et Métiers
M. **Jean-François Pradat-Peyre**, Professeur de l'Université Paris 10

LiP6 – Laboratoire d'Informatique de l'Université Pierre et Marie Curie

Table des matières

II Model checking parallèle et réparti

Résumé

Cette thèse s'inscrit dans le cadre de la vérification automatique de programmes concurrents basée sur un modèle formel intermédiaire : les réseaux de Petri colorés de haut-niveau.

Nous nous attachons à combattre le phénomène d'explosion combinatoire lié à l'exploration explicite de l'ensemble des entrelacements possibles du système. Pour cela, nous nous proposons de tirer profit de la quantité de mémoire et de la puissance de calcul offerte par un réseau local de machines travaillant de manière coopérative. Par le biais d'une analyse structurelle, nous cherchons à répartir efficacement le graphe d'accessibilité du système. Nous nous attachons ensuite à conserver l'efficacité des techniques visant à limiter l'explosion combinatoire dans cet environnement réparti en relâchant notamment les contraintes de cohérence sur l'exploration du graphe.

Nous avons alors validé ces approches à l'aide d'un vérifieur réparti et multithreadé dans lequel nous avons implémenté nos algorithmes.

Abstract

This thesis enters in the frame of the automatic verification of concurrent software based on an intermediary formal language : high-level colored Petri nets.

We particularly endeavor to tackle the combinatorial explosion phenomenon induced by exhaustive exploration of all possible inter leavings of a system. We focus on taking advantage of the amount of memory and computing resources provided by a local network of workstations working cooperatively. For more efficiency, distribution of the system accessibility graph is based upon a structural analysis of the model. We also aim at preserving in this distributed environment the other techniques that try to limit the combinatorial explosion. We especially relax constraints on the graph exploration consistency.

These approaches have then been validated and evaluated with our distributed and multithreaded model checker.

Remerciements

Je tiens tout d'abord à remercier Didier Buchs et Gaétan Hains pour avoir accepté de rapporter cette thèse dans des délais plus que limités. Je tiens également à remercier Franco Gasperoni, Claude Girault et Denis Poitrenaud pour avoir accepter de faire partie de mon jury.

Une grosse partie de cette thèse a été effectuée au sein du laboratoire CEDRIC du Conservatoire National des Arts et Métiers où j'ai pu côtoyer nombre de personnes qui m'ont aidé de diverses manières au cours de cette thèse. Je pense tout particulièrement à Olivier Alzéari, Philippe Auger, François Barthélémy, Joël Berthelin, Sami Evangelista, Gilles Lepage, Pierre Rousseau et Nicolas Trèves.

J'adresse également mes remerciements à tous les membres de l'équipe MoVe du LiP6 pour leur accueil : Fabrice Kordon, les thésards qui m'ont acceptés dans leur bureau, Alexandre Hamez, Alban Linard et Jean-Baptiste Voron, mais également Lom Hillah, Nicolas Gibelin et tous les autres que j'oublie forcément.

Cette thèse ne s'étant pas effectuée uniquement derrière un bureau dans un laboratoire, je souhaite également remercier mes parents ainsi que mes frères et sœur mais également plusieurs de mes amis et quelques élèves du CNAM avec lesquels j'ai passé de très bons samedis matins à faire de la programmation dans une très bonne ambiance. Parmi eux, je tiens tout spécialement à remercier Thais Botelho, Philippe Garilli, Thomas Martinez, Benjamin Mautret et Amel Zettili.

Et puis enfin, pour conclure, je souhaite adresser de profonds remerciements à Claude Kaiser et Jean-François Pradat-Peyre pour leur encadrement, leurs conseils et leur soutien tout au long de ces années.

Introduction

1 Le model checking

Cette section a pour objet de rappeler le principe du model checking et de sa mise en oeuvre : les algorithmes et techniques utilisés, les outils...

1.1 Principe général

Le model checking[1] est une technique de vérification basée sur l'exploration exhaustive de toutes les configurations, ou états, du système[2] à la recherche de comportements déviants de la spécification. Ces comportements sont spécifiés à l'aide de propriétés exprimées sur le modèle.

Un model checker peut être vu comme une boîte noire qui reçoit en entrée un système ainsi qu'une propriété exprimée sur ce système et produit en retour une réponse indiquant si la propriété est vérifiée ou non.

Les algorithmes mis en œuvre incluent généralement une construction de l'*espace d'état* du système puis un parcours de cet espace à la recherche d'erreurs. Cet espace d'état est un graphe dirigé qui décrit toutes les évolutions possibles du système. Les nœuds de ce graphe sont les états du système et les arcs représentent les transitions entre ces états. Ainsi, si le système étudié est un programme, un nœud correspondra à un état de la mémoire (valeurs des variables, compteurs ordinaux des processus, ...), et les transitions seront les instructions du programme. Le model checker peut aussi détecter ces erreurs en même temps qu'il construit l'espace d'état et s'arrêter si une telle erreur a été détectée. On parle alors de vérification à la volée.

1.2 Comparaison aux autres techniques de vérification

Un avantage majeur du model checking réside dans la précision de la réponse obtenue. Un model checker répondra en indiquant si la propriété est vérifiée ou non alors que les techniques de test et d'interprétation abstraite auront des réponses plus floues et ne permettront pas d'affirmer avec certitude que le système est correct.
De plus, cette procédure est totalement automatique : la tâche de l'utilisateur se résume à fournir au model checker un modèle ainsi que la propriété à vérifier. Notons toutefois que certains algorithmes ou optimisations peuvent nécessiter des informations fournies par l'utilisateur.
Enfin, une propriété appréciable d'un model checker est qu'il fournit un contre-exemple à l'utilisateur en cas d'erreur dans le modèle. Ce contre-exemple est une séquence de transitions dans l'espace d'état qui infirme la propriété spécifiée. Cette trace d'exécution est un outil précieux pour corriger le problème.

Cependant, le model checking souffre de deux limitations importantes.
Tout d'abord la plupart des algorithmes de model checking supposent que l'espace d'état du système étudié est fini. En effet, même le problème simple de l'accessibilité qui consiste à déterminer si

[1] Contrairement à une croyance largement répandue, le terme *model checking* ne signifie pas "vérifier que le modèle est correct" mais plutôt "vérifier que le système est un modèle d'une formule de logique".

[2] Le mot système prend ici un sens assez large puisque le model checking peut être appliqué à un modèle, p.ex., un réseau de Petri, un programme ou encore un composant matériel.

un état est accessible à partir de l'état initial du système par une quelconque séquence de transitions est indécidable pour une machine de Turing (ou tout formalisme ayant un pouvoir d'expression équivalent, p.ex., les réseaux de Petri avec arcs inhibiteurs) en raison de la possibilité de générer un espace d'état infini. Ainsi, si l'on souhaite vérifier un programme à l'aide d'un model checker il faudra s'assurer que celui-ci consomme une quantité de mémoire finie. Par conséquent, le model checking est théoriquement inapplicable à des programmes qui allouent dynamiquement de la mémoire ou qui contiennent des appels récursifs.
De plus, même si l'espace d'état du système est fini, il peut être si grand qu'il ne peut pas être exploré par les algorithmes faute de ressources (temps et mémoire). Ce problème bien connu est désigné sous le terme d'*explosion combinatoire* du nombre d'états. Deux facteurs sont principalement à la source de ce phénomène : la représentation de l'entrelacement des processus et des données. Prenons un exemple afin d'illustrer ce problème. Soit un système $\mathcal{S}$ composé de p processus indépendants, i.e., qui ne se synchronisent pas. Si chacun de ces processus peut être dans k états différents alors la taille de l'espace d'état de $\mathcal{S}$ sera de k^p. Cette taille croît donc exponentiellement par rapport au nombre de processus ce qui rend le problème du model checking très difficile même pour des petites valeurs de k et p. De plus k est fortement lié aux données manipulées par les processus. Ainsi, si chaque processus possède une variable entière codée sur 16 bits, k pourra potentiellement être de l'ordre de 2^{16}.

1.3 Model checking explicite et symbolique

Il existe deux grandes approches au problème du model checking : l'approche explicite et l'approche symbolique.

La première est basée sur une représentation explicite de l'espace d'état : chaque état est présent explicitement en mémoire dans une structure de données utilisée pour stocker l'espace d'état du système, p.ex., une table de hachage. La construction de cet espace s'effectue par un parcours de graphe. La figure 1 présente un algorithme de construction explicite. Celui-ci manipule deux ensembles : *Reach*, l'ensemble des états rencontrés qui est initialisé à l'état initial du système s_0 et *New*, le sous-ensemble des états de *Reach* qui n'ont pas encore été visités, i.e., dont les successeurs n'ont pas été générés. A chaque étape, l'algorithme sélectionne et retire de l'ensemble New un état (ligne 4 et 5) puis parcourt tous ses successeurs. Ceux qui ne sont pas dans l'espace d'état y sont insérés (ligne 7 à 10). L'algorithme se termine lorsqu'un point fixe a été atteint et que tous les états ont été visités.
Notons que selon l'implantation de la structure *New*, l'algorithme procédera en largeur (structure FIFO) ou en profondeur (structure LIFO).

```
procedure explore_explicit ()
1   New ← {s0}
2   Reach ← New
3   while New ≠ ∅ do
4     let s ∈ New
5     New ← New \ {s}
6     for s' ∈ Successor(s) do
7       if s' ∉ Reach then
8         New ← New ∪ {s'}
9         Reach ← Reach ∪ {s'}
10      end if
11    end for
12  end while
```

FIG. 1 – Construction explicite de l'espace d'état

L'approche symbolique, apparue plus tard avec les travaux de Kenneth McMillan [68], se dis-

tingue par les structures de données qu'elle utilise et les algorithmes qu'elle met en œuvre. Tout d'abord l'espace d'état n'est plus représenté explicitement, mais symboliquement, p.ex., à l'aide d'une formule de logique. Cela se traduit dans l'implantation par l'utilisation de structure de données compactes comme les diagrammes de décision binaire (ou *BDD* [66]) pour représenter des ensembles d'états. Un BDD est une forme d'arbre binaire qui permet de représenter efficacement des fonctions booléennes en permettant un partage fort entre les nœuds de l'arbre.
Les algorithmes symboliques de construction et d'exploration de l'espace d'état sont fondamentalement différents des algorithmes explicites. La figure 2 en présente un exemple. La différence majeure réside dans le fait que les structures de données symboliques permettent de traiter simultanément non pas un état mais un ensemble d'états. Ainsi, à chaque étape, l'algorithme peut calculer l'ensemble des nouveaux états accessibles à partir de l'ensemble *New* en une seule opération (ligne 4). De même, l'ajout des nouveaux états à l'espace d'état *Reach* est implanté par la disjonction de deux diagrammes de décision (ligne 5). Toutes ces opérations (calcul de $Successor(New)$, différence et disjonction de diagrammes de décision) ont des complexités linéaires par rapport à la taille des diagrammes de décision impliqués, et cette taille peut être bien inférieure au nombre d'états représentés (exponentiellement plus petite dans le meilleur des cas).
Il est à noter que les algorithmes symboliques procèdent implicitement par un parcours en largeur. L'ensemble New est initialisé à l'état initial s_0. A la première itération il contient tous les successeurs de l'état initial, i.e., ceux à la profondeur 1. A la seconde itération il contient tous les successeurs des états à la profondeur 1, et ainsi de suite. Le nombre d'itérations effectuées sera donc égal au diamètre de l'espace d'état. Il peut donc être problématique d'appliquer l'approche symbolique aux problèmes pour lesquels il est plus naturel de procéder autrement, p.ex., la vérification de propriétés LTL (voir la suite de cette introduction).

```
procedure explore_symbolic ()
1   New ← {s0}
2   Reach ← New
3   while New ≠ ∅ do
4      New ← Successor(New) \ Reach
5      Reach ← Reach ∪ New
6   end while
```

FIG. 2 – Construction symbolique de l'espace d'état

La mémoire est une ressource critique pour les model checkers explicites. Puisque chaque état est représenté explicitement en mémoire, il est difficile d'analyser des systèmes ayant plus de 10^6-10^7 états.
Avec l'approche symbolique, il est possible d'énumérer bien plus d'états : la taille d'un BDD (son nombre de noeuds) peut être exponentiellement plus petite que la taille de l'espace d'état qu'il représente. Cette approche a déjà été appliquée avec succès à la validation de systèmes de 10^{20} [67] états et plus récemment à des systèmes de 10^{150} états [138]. En pratique, les choses ne se passent pas toujours aussi bien, et il se peut que la taille du BDD explose elle aussi, i.e., que l'espace d'état ne puisse pas être efficacement représenté par une formule de logique.
En conclusion, ces deux approches sont complémentaires bien plus que concurrentes : il existe des systèmes pour lesquels l'une des approches est adaptée mais pas l'autre, et inversement.

1.4 Spin et SMV : deux outils de model checking pionniers

Les recherches menées dans le domaine du model checking ont été concrétisées par le développement de nombreux outils. Leurs domaines d'application sont variables : certains sont utilisés dans un cadre purement académique et d'autres, plus matures, ont permis de valider des systèmes réels soumis par des industriels.
La base de données Yahoda [137] recense quelques-uns de ces model checkers. Nous avons choisi de présenter les outils Spin et SMV qui sont deux outils pionniers dans leurs domaines.

Spin (Simple Promela Interpreter, [35]) est l'un des premiers outils de model checking : la première version publique remonte à 1991. C'est actuellement le model checker explicite de référence. Son formalisme d'entrée, le langage Promela (Process Meta-Language), permet de décrire des systèmes composés de processus communiquants à l'aide de canaux de communication ou de variables partagées. Il s'inspire à la fois du langage de commandes gardées de Dijkstra et de l'algèbre de processus CSP. Depuis sa version initiale, Spin s'est enrichi de nombreuses algorithmes et techniques de réduction qui expliquent son efficacité et sa popularité : vérification de propriété LTL, réduction par ordre partiel, représentation de l'espace d'état à l'aide d'un automate minimal, slicing, compression d'état ... Côté applications, Spin a permis de vérifier de nombreux systèmes industriels dans les domaines des télécommunications et de l'aérospatiale, entre autres. En raison de sa popularité, plusieurs variations de l'outil sont nées :

- dSpin [109] étend le langage Promela avec les éléments dynamiques des langages de haut niveau (allocation dynamique de mémoire, fonctions récursives, pointeurs, ...)
- HSF-Spin [110] intègre des algorithmes de recherche heuristique pour trouver les erreurs plus rapidement.

Depuis 1995, le workshop Spin contribue à l'amélioration de l'outil. Il réunit tous les ans chercheurs et utilisateurs afin de présenter des recherches et applications autour de Spin ou sur le thème plus général du model checking.

L'outil **SMV** (Symbolic Model Verifier, [120]), développé à l'université Carnegie Mellon, est un model checker symbolique basé sur les diagrammes de décision binaire. La première version de l'outil date de 1992. Son langage de description permet de modéliser des réseaux d'automates communiquants sur lesquels SMV peut vérifier des propriétés CTL. Cet outil tire sa popularité de son statut de premier model checker symbolique capable de vérifier des systèmes de taille très importante. Il est principalement utilisé pour la vérification de composants matériel. SMV n'est actuellement plus développé et il a laissé sa place à NuSMV [108], une réimplémentation de l'outil qui offre une architecture ouverte qui peut servir de noyau à d'autres outils de vérification.

2 Le model checking de programmes

Le model checking a longtemps été utilisé comme moyen de vérification des spécifications formelles plutôt que des implémentations. La raison principale est que les langages de programmation sont trop complexes et de trop bas niveau pour être vérifiés efficacement. Les langages formels manipulent des objets mathématiques comme des ensembles ou des fonctions alors que les programmes manipulent des bits, des tableaux ... Une autre difficulté réside dans le niveau d'abstraction employé. Lorsque l'on vérifie un modèle, on vérifie une abstraction du programme. Le concepteur de ce modèle peut alors volontairement mettre de côté tous les détails d'implémentation qui lui paraissent superflus pour vérifier que son modèle est valide vis à vis de la spécification. Le programme lui ne contient aucune abstraction ce qui complexifie la tâche du model checker.
Cependant, le model checking de programmes est une idée qui a émergé depuis la fin des années 1990 et des outils, souvent expérimentaux, sont apparus. Il y a deux approches possibles au problème : la vérification par abstraction - vérification, et la vérification directe du programme. Dans la suite de cette section, nous en présentons les principes ainsi que quelques outils représentatifs.

2.1 L'approche par traduction - vérification

Cette approche consiste à vérifier un modèle formel du programme plutôt que le programme lui-même. Le modèle doit être construit manuellement ou généré automatiquement par une traduction du code source du programme puis vérifié à l'aide d'un model checker "classique", p.ex., Spin ou SMV. Il est alors possible de bénéficier de l'efficacité de ces outils et des techniques de réduction qu'ils implémentent, ce qui est un avantage considérable. De plus, il est possible d'abstraire le programme durant la construction du modèle et de retirer les éléments qui pourraient complexifier

la vérification. De fait, ces abstractions sont généralement nécessaires en raison du fossé sémantique qui existe entre les langages de programmation et les langages de description des model checkers. Certaines constructions comme l'allocation dynamique de mémoire, les fonctions récursives, ou le ramasse-miette ne peuvent souvent pas être modélisées de manière satisfaisante dans le langage cible. Deux choix s'offrent alors au concepteur de l'outil : ne traiter qu'un sous-ensemble du langage de programmation qui inclut les éléments pouvant être traités ou employer des abstractions. Dans le second cas, le programme n'est plus réellement vérifié, et il est possible que l'outil signale des erreurs qui apparaissent dans le modèle mais pas dans le programme (ou, plus grave encore, que l'outil ne trouve pas des erreurs existantes). On parle alors de faux contre-exemples (*spurious counter-example*). Une tâche, parfois lourde, incombe alors à l'utilisateur qui doit examiner les erreurs signalées par l'outil afin de vérifier si elles sont réelles ou dues à une abstraction exagérée. Enfin, en cas d'erreur, le contre-exemple produit par le model checker est exprimé sur le modèle formel. L'outil doit alors le ré-exprimer sur le programme, opération qui n'est pas toujours facile.

a. Le paradigme CEGAR [99]

L'approche CEGAR (acronyme de *Counter Example Guided Abstract Refinement*) est implantée dans de nombreux outils fonctionnant par traduction - vérification. Il décompose le problème de la vérification en quatre étapes selon un procédé itératif schématisé par la figure 3.
Initialement, l'utilisateur soumet à l'outil un programme ainsi qu'une formule exprimée sur celui-ci, p.ex., un invariant. La première étape consiste à abstraire le programme et à le traduire dans le langage d'un model checker. Au début du processus, le modèle doit être le plus abstrait possible, mais doit suffisamment "simuler" le programme de telle sorte que toute exécution du programme se retrouve dans le modèle abstrait.
Un model checker permet ensuite de trouver des erreurs dans le modèle. Si le modèle ne contient aucune erreur, l'abstraction nous garantit qu'aucune erreur n'est présente dans le programme. Le processus peut alors s'arrêter, et l'utilisateur reçoit la confirmation que la formule est effectivement vérifiée.
Il se peut aussi que la formule ne soit pas vérifiée par le modèle. Dans ce cas, il convient de vérifier si l'erreur peut être reproduite sur le programme, i.e., si elle n'est pas due à une "sur-abstraction". Le problème de déterminer si un contre-exemple est faux étant indécidable, cette opération requiert souvent l'assistance de l'utilisateur. Si l'erreur est bien présente dans le programme, alors elle est directement renvoyée à l'utilisateur. Dans le cas contraire, l'erreur trouvée n'est que la conséquence d'une mauvaise abstraction du programme. En se basant sur ce faux contre-exemple, l'outil raffine et améliore l'abstraction afin qu'elle soit plus proche du programme initial. Le processus peut reprendre sur le nouveau modèle abstrait. Il s'arrêtera lorsque l'abstraction sera assez bonne pour valider la formule ou pour trouver une erreur reproductible sur le programme.

b. Quelques model checkers basés sur cette approche

Les deux premiers outils présentés suivent l'approche CEGAR. Ils sont tous les deux basés sur des model checkers symboliques utilisant des diagrammes de décision binaire.

L'outil **Blast** (*Berkeley Lazy Abstraction Software verification Tool* [117]) vérifie les propriétés de sûreté, i.e., déterminer si un état erroné est accessible, des programmes C. Sa technique d' "abstraction fainéante" (*lazy abstraction* [100]) lui permet de réutiliser les abstractions durant les itérations successives et de ne pas réexplorer, durant l'étape de model checking, les parties de l'espace d'état qui, d'après une abstraction précédente, ne contiennent pas d'erreur. La version actuelle de l'outil ne prend pas en charge les pointeurs de fonctions et les appels récursifs.

Slam [106] est un outil développé par les laboratoires Microsoft. Il s'applique à des programmes C séquentiels et permet de vérifier qu'un programme est un "bon client" d'une bibliothèque de programmes, p.ex., le programme n'appelle jamais la fonction f avec le paramètre i égal à 0, tout

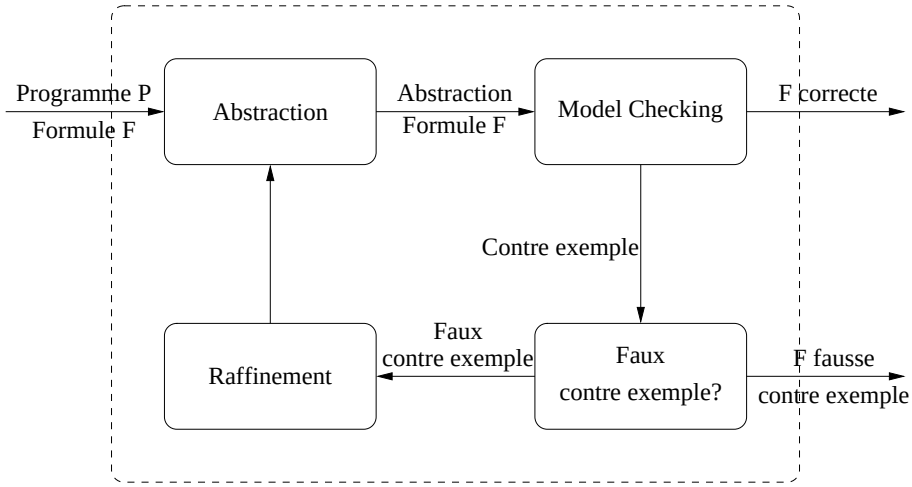


FIG. 3 – Le paradigme CEGAR

appel à la fonction *lock* est suivi d'un appel à la fonction *unLock*, ... Slam implante des techniques d'analyse statique qui lui permettent de limiter efficacement les bruits (les fausses erreurs). En pratique, il a déjà été utilisé pour analyser des programmes de plus de 10 000 lignes de code et a trouvé des erreurs dans des pilotes du système Windows.

Les deux outils suivants ne sont pas à proprement parler des model checkers de programmes, mais ils disposent de langages de spécification assez proches des langages de programmation pour permettre une traduction avec un minimum d'abstractions. Ils peuvent donc aisément s'intégrer à des outils de vérification de programmes.

Spin a été initialement conçu pour vérifier des protocoles, mais il est aussi adapté à la vérification de programmes en raison de son langage de spécification proche du C. De plus, il est possible, depuis la version 4.0, d'intégrer du code C dans les modèles. Cette fonctionnalité facilite considérablement le passage d'un programme C à un modèle Promela. En pratique, Spin a permis de vérifier de nombreuses applications dont un commutateur téléphonique développé par Lucent Technologies. Dans le cadre de ce projet, seize machines ont été quotidiennement utilisées durant plusieurs mois afin d'effectuer des vérifications avant la mise sur le marché du commutateur. C'est probablement l'exemple d'application du model checking de programmes le plus probant à ce jour.

Zing [105] est un model checker explicite récent développé par les laboratoires Microsoft. Son langage de spécification inclut la plupart des éléments des langages de programmation de haut niveau : fonctions et procédures, exceptions, allocation dynamique, appels récursifs, types de données de haut niveau, classes (bien que l'héritage ne soit pas possible). Les processus peuvent se synchroniser par l'envoi de messages ou par l'utilisation de variables partagées. Zing propose plusieurs techniques pour lutter contre l'explosion combinatoire : réduction par ordre partiel, prise en compte des symétries, vérification compositionnelle,... Cet outil est encore au stade de prototype mais il a tout de même permis de trouver des erreurs dans des pilotes du système Windows.

2.2 L'approche par vérification directe du programme

Cette seconde approche du problème consiste à appliquer le model checking directement sur le programme par un mécanisme d'exécution symbolique. Elle présente un avantage considérable par rapport à l'approche par traduction - vérification : elle permet de trouver de vrais bogues. En effet, toute erreur signalée par l'outil est nécessairement reproductible par le programme puisque

c'est le programme et non une abstraction du programme qui est vérifiée. L'utilisateur est donc déchargée de la tâche pénible qui consiste à examiner les potentielles erreurs détectées par l'outil pour déterminer si elles sont réelles. De plus, les contre-exemples trouvés par le model checker sont exprimées directement sur le programme et non sur le modèle formel comme c'est le cas avec plusieurs outils appartenant à la première famille. Ceci constitue une aide appréciable pour l'utilisateur. En contrepartie, les model checkers de cette famille sont particulièrement sensibles au problème d'explosion combinatoire puisqu'ils ne réalisent aucune abstraction.

a. Quelques model checkers basés sur cette approche

Verisoft [114, 115] est le premier model checker de cette famille. Il est développé par Patrice Godefroid des laboratoires Bells qui est un des "pères" des techniques de réduction par ordre partiel. Verisoft peut vérifier différents types de propriétés sur des programmes concurrents écrits en C ou C++ : l'absence d'état mort (i.e., sans possibilité d'évolution) ou de divergence (i.e., un processus exécute indéfiniment une portion de code inutile, demander une ressource qu'il n'obtient jamais par exemple) ou encore des assertions.
L'exécution du système est contrôlée par un processus additionnel inséré par l'outil. Ce processus, appelé ordonnanceur, a un rôle crucial puisqu'il va guider le programme dans son espace d'état. Il peut suspendre un processus et donner la main à un autre processus à certains états du système jugés stratégiques. Pour déterminer ces états, Verisoft classe les différentes instructions du programme en deux familles : les instructions visibles et invisibles. Ces dernières correspondent à des transitions inintéressantes du point de vue du système global. Cette classification est établie statiquement à l'aide de la théorie des réductions par ordre partiel (voir le chapitre 6 pour de plus amples détails). L'écriture dans une variable locale sera par exemple considérée comme invisible. Une synchronisation inter-processus, p.ex., la prise d'un verrou, sera par contre une instruction visible. Le principe de l'algorithme de l'ordonnanceur est de laisser un processus s'exécuter tant qu'il n'exécute que des instructions invisibles. Lorsque tous les processus sont bloqués sur des instructions visibles l'ordonnanceur considère tous les entrelacements possibles et donne successivement la main à tous les processus.
Un principe fondateur de l'outil est que les états des programmes sont trop complexes pour être représentés en mémoire de manière canonique et non ambiguë comme le font les model checkers explicites, p.ex., à l'aide d'un vecteur de bits. L'algorithme de recherche de Verisoft ne garde donc pas les états en mémoire. Il est qualifié de *stateless* (sans état). Deux problèmes se posent alors :

- assurer la terminaison de l'algorithme (si l'espace d'état contient des cycles)
- éviter les visites répétées du même état qui pénaliseraient fortement les performances de l'algorithme

Les deux réponses apportées par l'outil à ses problèmes sont les suivantes. D'une part, la recherche est limitée à une profondeur maximale afin de garantir la terminaison. Cette profondeur est spécifiée par l'utilisateur. D'autre part, les techniques de réduction par ordre partiel permettent en partie de résoudre le second problème en limitant les visites répétées d'un même état. Ce choix de conception a donc un coût en terme de complétude (possibilité de manquer un état erroné se trouvant à une profondeur supérieure à la profondeur maximale) et de performances (visites redondantes d'un même état).

Java Pathfinder (JPF, [116, 122]) est un model checker pour programmes Java concurrents. Son nom provient du robot *Mars Pathfinder* envoyé sur la planète Mars et dont le fonctionnement s'arrêta brutalement en raison d'une défaillance logicielle : un erreur de conception de l'ordonnanceur temps réel aboutit à une inversion de priorité des tâches. Dans sa première version, JPF n'était pas un model checker mais un outil de traduction de programmes Java en modèles Promela, le langage de l'outil Spin. Il faisait alors partie de la première partie de model checkers présentée précédemment. Depuis sa seconde version, JPF utilise une machine virtuelle spécifique pour vérifier directement le byte-code Java compilé. Il ressemble donc fortement au précurseur Verisoft. Cependant, il se distingue de celui-ci par ses algorithmes de recherche plus "classiques", i.e., avec

représentation et stockage des états du programme. JPF fournit aussi des algorithmes de recherche heuristique qui permettent d'accélérer la détection des erreurs.

3 Combattre l'explosion combinatoire

En raison du phénomène d'explosion combinatoire, le model checking a longtemps été considéré comme un outil immature et inadéquat pour les systèmes industriels. Cependant, nombre de chercheurs se sont attelés au problème durant ces trois dernières décennies et ont conçu des techniques permettant de le combattre efficacement. En conséquence, le model checking s'est largement répandu dans l'industrie et est maintenant couramment utilisé pour vérifier des protocoles ou des composants matériel. Plus récemment, le model checking de programmes a émergé comme un nouveau thème de recherche. Son utilisation dans ce cadre particulier soulève des difficultés supplémentaires dont nous discuterons par la suite.
Nous nous proposons de dresser un aperçu des techniques majeures utilisées pour combattre l'explosion combinatoire.

Les techniques d'**ordre partiel** s'attaquent à l'une des sources principales du problème : l'exécution concurrente de plusieurs processus. Elles sont basées sur l'observation suivante : en raison de la sémantique d'entrelacement des systèmes concurrents, plusieurs exécutions différentes peuvent avoir le même effet sur le système et n'être qu'une permutation d'une même séquence. Ainsi, une manière efficace de réduire l'explosion d'états serait de n'explorer qu'une seule, ou quelques-unes de ces séquences et d'ignorer toutes les autres permutations qui sont équivalentes aux séquences retenues. C'est pour cette raison que l'on emploie aussi le terme de model checking d'*ensembles représentatifs*.

Les systèmes informatiques présentent généralement un haut degré de **symétrie**. On trouve de telles symétries dans les mémoires, les caches, les protocoles. . ., et plus généralement dans tout système composé d'éléments répliqués. Or il est possible de tirer partie de ces symétries pour réduire le nombre d'états visités. Considérons par exemple un protocole avec N processus identiques. Puisque ces processus exécutent le même algorithme il n'est probablement pas nécessaire de se soucier de leur identité. Par conséquent, plutôt que de décrire un état du système de la manière suivante : "p_0 *et* p_1 *sont dans l'état* s *et* p_2 *est dans l'état* s'", on le décrira ainsi : "*deux processus sont dans l'état* s *et un processus est dans l'état* s'". Le principe de cette réduction est donc de grouper les états symétriques en classes d'équivalence et de ne visiter qu'un état par classe. Toute la difficulté est de déterminer quand deux états sont équivalents. Ce problème étant très difficile (au moins NP) on s'aide plutôt de la structure du système pour déterminer des conditions suffisantes.

Les techniques **symboliques** représentent l'espace d'état à l'aide d'une structure de données qui permet un partage de données fort entre différents états. On utilise principalement les diagrammes de décision binaire qui sont une représentation canonique et pour lesquels il existe des algorithmes efficaces. En outre, cette structure permet d'accélérer considérablement la recherche. Il devient alors possible d'explorer des espaces d'état de taille gigantesque, 10^{20} par exemple. Nous reviendrons plus tard dans cette introduction sur cette technique.

Nombre de techniques d'**abstraction** peuvent être appliquées non pas durant la recherche mais directement sur le modèle. Le principe général est de transformer le modèle initial en un modèle plus simple (au sens de la vérification) et, si possible, équivalent par rapport à la propriété spécifiée. Nous citerons, entre autres, la réduction de réseau de Petri et la découpe de programme qui consiste à retirer les éléments superflus d'un programme qui n'influent pas sur la propriété à vérifier.

Cette liste est loin d'être exhaustive et la littérature foisonne d'algorithmes et de techniques conçus pour relever ce défi. Nous citerons en vrac : la vérification modulaire, la compression d'état (p.ex. en utilisant un codage de Huffman). Pour de plus amples informations, le lecteur pourra

consulter [30], ouvrage consacré au model checking, ou encore [144] qui dresse un aperçu plus poussé des techniques de réduction.
Il est important de noter que l'efficacité de ces méthodes dépend pour beaucoup du système sur lequel elles sont appliquées et qu'il est bien souvent impossible de prévoir comment elles se comporteront. Ainsi, une technique pourra se révéler inadaptée pour tel modèle mais réduira exponentiellement l'espace d'état de tel autre modèle. L'utilisation conjointe de ces méthodes est donc indispensable pour pouvoir vérifier des systèmes industriels.

4 Répartition du processus de vérification

Les stratégies présentées dans la section précédentes ont toutes pour objectif de réduire la quantité de mémoire requise pour la validation d'un système donné. Soit en compressant la représentation des données, soit en utilisant notamment la structure du modèle afin de limiter la quantité de données à conserver.

Les réseaux de machines étant, depuis quelques décennies, extrêmement répandus, une autre approche totalement différente est apparue pour essayer de répondre au problème de l'explosion combinatoire en l'attaquant de manière plus frontale. Le principe est de tirer profit du regroupement de machines interconnectées en un réseau local afin de disposer de ressources mémoires beaucoup plus importantes.

Cette stratégie ne peut bien évidemment pas remplacer les précédentes et se donc d'être combinée avec elles. Le caractère réparti du parcours de l'espace pose alors quelques problèmes d'adaptation tant les méthodes séquentielles présentées dans la section précédente sont, pour certaines, étroitement liées au parcours – et surtout à son caractère cohérent – pour plus d'efficacité.

5 Conclusion

Malgré l'intérêt grandissant qu'il suscite dans la communauté scientifique le model checking de programmes est une technologie encore très jeune et difficilement applicable à des programmes réels. Les outils disponibles sont pour la plupart de nature académique et leur utilisation nécessite un haut niveau d'expertise et une connaissance profonde des formalismes utilisés, p.ex., la logique temporelle. A notre connaissance, ces outils n'ont jamais été utilisés par des non spécialistes dans des projets de grande envergure, alors que les outils de test et de preuve automatique se sont largement répandus et sont maintenant couramment utilisés.

Les deux approches de vérification présentées dans cette section sont radicalement différentes dans leurs philosophies. Il est important de noter que les outils qui suivent l'approche par traduction - vérification ne sont finalement pas stricto sensu des model checkers. En effet, en raison des abstractions qu'ils utilisent, si faibles soient elles, ils ne permettent pas de vérifier formellement des programmes mais plutôt de signaler des erreurs potentielles. De plus, comme nous l'avons déjà remarqué, ils sont loin d'être automatiques et requièrent généralement l'assistance de l'utilisateur. Il serait donc plus judicieux de les qualifier d'outils d'aide au débogage. Ils sont en cela fondamentalement différents des outils basés sur une vérification directe du code, p.ex., Verisoft et JPF : ces outils trouvent de vrais bogues. Patrice Godefroid fait part dans [139] de son étonnement à ce sujet : la plupart des outils fonctionnent par abstraction alors que ce qui importe réellement est de trouver de vrais bogues (*"The soundess of bugs is what matters"*). Il préfère ainsi sacrifier la complétude (*soundness*) à l'exactitude (*soundness of bugs*).

Pour conclure, nous pensons que les techniques de model checking ne devraient pas être utilisées une fois le programme entier écrit, mais devraient pleinement s'intégrer à l'implémentation et la guider selon un processus itératif. Un premier squelette du programme serait généré, si possible

automatiquement, à partir du modèle formel puis automatiquement vérifié. Puis le programme serait raffiné par itérations successives. A chaque itération, de nouveaux éléments seraient rajoutés au programme, et le model checker vérifierait de nouvelles propriétés et s'assurerait que les modifications apportées n'ont pas altéré les propriétés prouvées à des étapes précédentes. La nature des erreurs vérifiées changerait progressivement. Au début de l'implémentation, les erreurs seraient plutôt de nature conceptuelle, et deviendrait petit à petit des erreurs d'implémentation.
Cette approche présente deux avantages. D'une part, elle permet de trouver et de corriger les bogues plus tôt et limite ainsi leur coût. D'autre part, il se peut qu'à une certaine étape le model checker ne soit plus en mesure de vérifier le programme en raison du problème d'explosion combinatoire. Il est alors très appréciable pour l'utilisateur de savoir que son programme est valide jusqu'à un certain point et que certaines modifications sont potentiellement source d'erreurs. Cette information peut par exemple être utilisée pour définir des tests ultérieurs plus ciblés. De plus, il est toujours plus intéressant de recevoir comme réponse de la part du model checker "*la version N du programme est valide*" plutôt que "*j'ai essayé de vérifier le programme final mais je ne peux pas vous répondre faute de mémoire*".

6 Plan de la thèse

Cette thèse se place dans le contexte de la vérification automatique de programmes concurrents en se basant sur du model checking explicite par le biais d'un modèle formel intermédiaire : les réseaux de Petri. Nous aborderons plus spécifiquement les aspects parallèles et répartis du model checking explicite de réseaux de Petri. Bien que nous nous situions dans le cadre de la vérification de programmes concurrents, nos approches sont suffisamment générales pour être étendue au cadre, plus large, du model checking explicite.

Ce mémoire est divisé en trois parties, elle même découpées en différents chapitres.

La première partie a pour objectif de présenter les notions que nous utiliserons par la suite dans cette thèse. Le premier chapitre décrit notamment les aspects formels. Plusieurs définitions ainsi que les définitions relatives aux réseaux de Petri, ainsi qu'aux réseaux de Petri colorés dont ils sont une généralisation, peuvent être trouvées.
Le second chapitre, quant à lui, présente quelques notions relatives aux systèmes répartis et multiprocesseurs que nous utiliserons lors de cette thèse. Nous y décrirons les principales caractéristiques liées à ce type de systèmes.

La seconde partie est composée de 4 chapitres. Elle traite directement du model checking parallèle et réparti.
Le premier chapitre de cette partie présente l'adaptation de model checking à un environnement réparti. Nous y présenterons les méthodes utilisées pour partitionner les données sur le réseau le plus efficacement possible. Nous aborderons les mécanismes qui nous ont permis d'adapter les méthodes de compression de données à un environnement réparti.

Les méthodes permettant de combattre l'explosion combinatoire présentées dans 0.3 ralentissent toutes – plus ou moins – la vérification car elles introduisent un surcoût parfois non négligeable en terme de temps de calcul. Le quatrième chapitre présente donc comment tirer profit des machines multiprocesseurs, de plus en plus répandues, pour réduire ce surcoût. Nous verrons alors comment paralléliser au mieux (en utilisant des mécanismes de synchronisation adaptés) le processus de vérification pour obtenir des gains de temps intéressants. Nous verrons également quels sont les ajustements à effectuer pour adapter les algorithmes utilisés en séquentiel. Nous discuterons également des points communs et principales différences entre le model checking réparti et le model checking parallèle.
Le chapitre suivant présente une combinaison des solutions architecturales présentées dans les deux précédents chapitres. Cette combinaison vise à tirer profit des deux stratégies pour les res-

sources mémoires et de calcul. Notons que pour des raisons purement matérielles, les expérimentations de cette solution architecturale n'ont pu être extrêmement poussées et restent préliminaires. Elles permettent néanmoins de tirer de premières conclusions.

Le sixième chapitre – et dernier de la seconde partie – traite d'une méthode de réduction présentée dans la section 3 de cette introduction : les réductions d'ordre partiel. Cette technique de réductions qui permet de limiter la recherche à un sous-ensemble pertinent des configurations possibles du système est particulièrement efficace. Malheureusement, pour être la plus efficace possible, cette approche repose très fortement sur la cohérence globale du parcours utilisé pour l'exploration des configurations possibles du système. Hors, dès lors que l'on effectue cette exploration *via* le model checking réparti ou parallèle, cette cohérence globale ne peut plus être assurée. Nous présentons donc dans ce chapitre un méthode permettant de relâcher en partie la nécessité de cohérence du parcours. Cette méthode sera tout d'abord présentée dans un environnement séquentiel dans lequel elle peut tout à fait fonctionner de manière très efficace. Puis nous verrons comment elle peut être adaptée simplement aux environnements répartis et parallèles.

Enfin, la dernière partie présente les outils utilisés pour la vérification de programmes concurrents. Dans le premier chapitre nous présenterons la plateforme Quasar qui permet la vérification de programme concurrent. Nous y verrons notamment comment un modèle comme les réseaux de Petri peut répondre au problème posé par les aspects dynamiques des langages de programmation.

Le dernier chapitre sera, quant à lui, consacré à l'outil Cyclades, le model checker parallèle et réparti développé lors de cette thèse. Nous présenterons rapidement ses principales caractéristiques et décrirons également un algorithme permettant l'analyse structurelle des réseaux de Petri colorés.

Nous finirons par effectuer une synthèse rapide des travaux effectués lors de cette thèse et présenterons quelques perspectives de recherche à plus ou moins long terme.

Première partie

Préliminaires

Chapitre

1

Réseaux de Petri

Sommaire

Nous donnons dans ce chapitre les notations et définitions nécessaires sur les réseaux de Petri et les réseaux de Petri colorés. La première section rappelle quelques notations usuelles.

1.1 Notations et conventions

Dans cette thèse, nous noterons :
- $\mathbb{N} = \{0, 1, \ldots\}$ l'ensemble des entiers naturels
- $\mathbb{N}^+ = \{1, \ldots\}$ l'ensemble des entiers naturels positifs
- $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ l'ensemble des entiers relatifs
- $\mathbb{Q}$ l'ensemble des rationnels
- $\mathbb{B} = \{true, false\}$ l'ensemble des booléens
- E^N l'ensemble des vecteurs de dimension N à valeur dans E. C'est l'ensemble des applications de N vers E. Un élément de $E^{N \times M}$ est appelé une matrice de dimension $N \times M$ à valeur dans E.
- ${}^tA$ la transposée de la matrice A. Si $A \in E^{N \times M}$ alors ${}^tA \in E^{M \times N}$ est définie par : ${}^tA(i, j) = A(j, i)$. Puisqu'un vecteur de E^N peut être considéré comme une matrice de $E^{N \times \{1\}}$, cette définition est aussi valable pour les vecteurs.

1.1.1 Multi-ensembles et ensembles des parties d'un ensemble

La définition et l'analyse des réseaux de Petri colorés, qui seront définis ultérieurement dans ce chapitre, sont basées sur la notion de multi-ensemble. Intuitivement un multi-ensemble est un ensemble dans lequel un élément peut apparaître plusieurs fois.

Définition 1.1 (Multi-ensemble)**.** *Soit E un ensemble. Un multi-ensemble sur E est une fonction de E vers $\mathbb{N}$. L'ensemble des multi-ensembles sur E est noté $Bag(E)$. Pour tout ensemble E, $\emptyset \in Bag(E)$ est le multi-ensemble vide défini par $\forall e \in E, \emptyset(e) = 0$. Soient m un multi-ensemble sur E et $e \in E$. $m(e)$ est appelé la multiplicité de e.*

Un multi-ensemble sera généralement noté comme une combinaison linéaire ou à l'aide de la notation ensembliste. Par exemple, le multi-ensemble m sur l'ensemble $\{a, b, c\}$ défini par $m(a) = 0, m(b) = 2, m(c) = 3$ pourra être noté $m = 2b + 3c$ ou encore $m = \{2b, 3c\}$.

Certaines opérations sur les multi-ensembles peuvent être définies.

Définition 1.2 (Opérations sur les multi-ensembles)**.** *Soient E un ensemble, $a \in Bag(E)$, $b \in Bag(E)$, et $\lambda \in \mathbb{N}$.*

$$
\begin{array}{lcl}
\lambda \cdot a & = & \sum_{x \in E} (\lambda \cdot a(x)) \cdot x \\
a + b & = & \sum_{x \in E} (a(x) + b(x)) \cdot x \\
a - b & = & \sum_{x \in E} \max(0, a(x) - b(x)) \cdot x \\
a \cap b & = & \sum_{x \in E} \min(a(x), b(x)) \cdot x \\
a \geq b & \Leftrightarrow & \forall x \in E, a(x) \geq b(x) \\
a > b & \Leftrightarrow & a \geq b \wedge \exists x \in E \mid a(x) > b(x)
\end{array}
$$

Nous noterons $\mathcal{P}(E)$ l'ensemble des parties de E. C'est l'ensemble des sous-ensembles de E.

Définition 1.3 (Ensemble des parties d'un ensemble)**.** *Soit E un ensemble. L'ensemble des parties de E est l'ensemble $\{P \mid P \subseteq E\}$. Il est noté $\mathcal{P}(E)$.*

1.1.2 Les séquences

Le terme de séquence de transitions sera fréquemment utilisé dans cette thèse. Une séquence sur un ensemble E est une suite, finie ou infinie, d'éléments de E.

Définition 1.4 (Séquence)**.** *Soit E un ensemble. Une séquence finie σ sur E est une fonction de $[1..|\sigma|]$ vers E. $|\sigma|$ est la longueur de la séquence. Une séquence infinie γ est une fonction de $\mathbb{N}^+$ vers E. L'ensemble des séquences finies (resp. infinies) sur E est noté E^* (resp. E^∞). La séquence vide est notée ϵ. Si $\sigma \in E^*$, et $\exists i$ tel que $\sigma(i) = e$ alors nous noterons $e \in \sigma$.*

L'opérateur "." est utilisé pour concaténer des séquences.

Définition 1.5 (Concaténation)**.** *Soient E un ensemble, $\sigma \in E^*$, et $\sigma \in E^* \cup E^\infty$. La concaténation de σ et σ' notée $\sigma.\sigma'$ est définie par : $\sigma.\sigma'(i) = \sigma(i)$ si $i \leq |\sigma|$, $\sigma'(i)$ sinon.*

La notation fonctionnelle sera rarement utilisée pour noter les séquences. Nous préférerons les noter à l'aide de l'opérateur de concaténation. Par exemple, la séquence $\sigma = a.b.c$ est définie par $|\sigma| = 3$, $\sigma(1) = a$, $\sigma(2) = b$, et $\sigma(3) = c$. De plus, si aucune ambiguïté n'est possible, le "." sera omis.

L'opérateur de projection permet de retirer d'une séquence sur E les éléments n'appartenant pas à un sous-ensemble de E.

Définition 1.6 (Projection)**.** *Soient E un ensemble, $\sigma \in E^* \cup E^\infty$ et $S \subseteq E$. La projection de σ sur S, noté $\Pi_S(\sigma)$, est définie récursivement par*

- *Si $\sigma = \epsilon$ alors $\Pi_S(\sigma) = \epsilon$*
- *Si $\sigma = e.\sigma'$, et $e \in S$ alors $\Pi_S(\sigma) = e.\Pi_S(\sigma')$*
- *Si $\sigma = e.\sigma'$, et $e \notin S$ alors $\Pi_S(\sigma) = \Pi_S(\sigma')$*

Par extension, si $\Sigma \in \mathcal{P}(E^* \cup E^\infty)$, $\Pi_S(\Sigma) = \{\sigma \mid \exists \sigma' \in \Sigma \mid \Pi_S(\sigma') = \sigma\}$

L'ensemble des préfixes d'une séquence σ est noté $Pref(\sigma)$.

Définition 1.7 (Ensemble des préfixes d'une séquence)**.** *Soient E un ensemble et $\sigma \in \sigma^* \cup \sigma^\infty$. L'ensemble des préfixes de σ est l'ensemble $\{\sigma' \mid \exists \sigma'', \sigma = \sigma'.\sigma''\}$. Il est noté $Pref(\sigma)$.*

1.2 Les réseaux de Petri

Les réseaux de Petri sont un outil mathématique et graphique adapté pour la représentation de systèmes concurrents. Nous rappelons dans cette section quelques définitions et notations sur ces réseaux. Une définition plus complète pourra être trouvée dans [9] qui est un "tutorial" sur les réseaux de Petri ou encore dans [2] qui présente les réseaux de Petri ainsi que diverses extensions au modèle et techniques d'analyse. La lecture de [3], qui rappelle de nombreux résultats sur la décidabilité de certaines propriétés "classiques", p.ex., l'accessibilité, le caractère borné, des réseaux de Petri, est aussi recommandée.

Un réseau de Petri est un graphe dirigé possédant deux types de nœuds : les places qui représentent l'état du système, et les transitions qui représente la dynamique du système. Les connections entre ces nœuds sont données par les matrices d'incidence.

Définition 1.8 (Réseau de Petri)**.** *Un réseau de Petri est tuple $\langle P, T, W^-, W^+\rangle$ où P est un ensemble fini de places ; T est un ensemble fini de transitions tel que $P \cap T = \emptyset$; W^- (resp. W^+) est une fonction d'incidence arrière (avant), de domaine $P \times T$ et de co-domaine $\mathbb{N}$. La fonction W de $P \times T$ vers $\mathbb{N}$ est définie par $W(p,t) = W^+(p,t) - W^-(p,t)$.*

A chaque fois qu'un réseau de Petri N, N', ou N_u sera défini, le tuple $\langle P, T, W^-, W^+\rangle$, $\langle P', T', W^{-\prime}, W^{+\prime}\rangle$, $\langle P_u, T_u, W^-{}_u, W^+{}_u\rangle$ qui le définit sera lui aussi implicitement défini.

Les matrices d'incidence d'un réseau de Petri peuvent être étendues à des matrices indicées par $P \times T^*$ de la manière suivante :

Définition 1.9. *Soient N un réseau de Petri, $p \in P$, et $\sigma \in T^*$.*
- *si $\sigma = \epsilon$ alors $W(p,\sigma) = W^-(p,\sigma) = W^+(p,\sigma) = 0$*
- *si $\sigma = \sigma'.t$ alors*
  - $W(p,\sigma) = W(p,\sigma') + W(p,t)$
  - $W^-(p,\sigma) = W^-(p,\sigma') + \max(0, W^-(p,t) - W^+(p,\sigma'))$
  - $W^+(p,\sigma) = W(p,\sigma) + W^-(p,\sigma)$

Pour représenter un réseau de Petri on préfère généralement la représentation graphique à la représentation formelle. Les places sont représentées par des cercles, et les transitions par des barres. Un arc allant d'une place p vers une transition t (resp. d'une transition t vers une place p) et étiqueté par un entier n signifie que $W^-(p,t) = n$ (resp. $W^+(p,t) = n$). L'étiquette par défaut des arcs est 1 et les arcs étiquetés par 0 sont retirés du graphique.

Une notation fréquemment utilisée est celle d'ensemble de sorties (ou d'entrées).

Définition 1.10 (Ensemble de sorties et d'entrées)**.** *Soient N un réseau de Petri, $p \in P$ et $t \in T$. Nous noterons*
- $^\bullet p = \{t \in T | W^+(p,t) > 0\}$*, l'ensemble des entrées de p*
- $p^\bullet = \{t \in T | W^-(p,t) > 0\}$*, l'ensemble des sorties de p*
- $^\bullet t = \{p \in P | W^-(p,t) > 0\}$*, l'ensemble des entrées de t*
- $t^\bullet = \{p \in P | W^+(p,t) > 0\}$*, l'ensemble des sorties de t*

Ces notations peuvent être étendues de manière directe à des ensembles de places ou de transitions. Par exemple, si $P' \subseteq P$, $^\bullet P' = \cup_{p \in P'} {}^\bullet p$.

Un marquage d'un réseau de Petri associe un nombre de jetons à toute place du réseau. Ces jetons sont représentés graphiquement par des points présents dans la place concernée.

Définition 1.11 (Marquage)**.** *Soit N un réseau de Petri. Un marquage m de N est une fonction de P vers $\mathbb{N}$. L'ensemble des marquages de N est noté $\mathcal{M}_N$.*

L'ensemble $\mathcal{M}_N$ pourra aussi être noté $\mathcal{M}$ si il n'y a pas d'ambiguïté.

Un marquage initial est généralement associé à un réseau de Petri. Celui-ci spécifie la distribution initiale des jetons dans les places du réseau. C'est l'état initial du système avant tout franchissement de transition.

Définition 1.12 (Réseau de Petri marqué)**.** *Un Réseau de Petri marqué est un couple (N, m_0) avec N un réseau de Petri et m_0 est un élément de $\mathcal{M}_N$.*

De même que la définition d'un réseau de Petri N implique la définition implicite du tuple $\langle P, T, W^-, W^+\rangle$, la définition d'un réseau de Petri marqué implique la définition implicite de son marquage initial m_0.

La sémantique d'un réseau de Petri est définie par la règle de franchissement, ou règle de tir. Une transition t est dite franchissable, permise, ou encore sensibilisée, si toutes ses places en entrée sont marquées en quantité suffisante. Le franchissement, ou tir, ou encore exécution, de t retire des jetons de ses places en entrée et en dépose dans ses places en sortie.

Définition 1.13 (Règle de tir)**.** *Soient N un réseau de Petri, $m \in \mathcal{M}_N$, et $t \in T$. La transition t est franchissable en m si et seulement si : $\forall p \in P, m(p) \geq W^-(p,t)$. Le franchissement de t en m mène à un marquage m' defini par $\forall p \in P, m'(p) = m(p) + W(p,t)$. Nous noterons :*
– $m[t\rangle$ la franchissabilité de t en m
– $m[t\rangle m'$ (ou $Succ(m,t) = m'$) si le franchissement de t en m mène au marquage m'
La séquence $\sigma \in T^* \cup T^\infty$ est franchissable en m si et seulement si $\sigma = \epsilon$ ou $\sigma = t.\sigma' \wedge m[t\rangle m' \wedge m'[\sigma'\rangle$. Nous notons $m[\sigma\rangle$ la franchissabilité de σ en m.
Le franchissement d'une séquence $\sigma \in T^*$ en m mène à un marquage m' défini par $\forall p, m'(p) = m(p) + \sum_{i=1}^{|\sigma|} W(p, \sigma(i))$. Le franchissement de σ en m est noté $m[\sigma\rangle m'$.

Certains marquages sont caractérisés par le fait que leur ensemble de franchissabilité est vide. Un tel marquage est dit *mort*. Ils résultent généralement d'une erreur de conception dans le modèle, ou d'un état de terminaison dans le système.

Définition 1.14 (Marquage mort)**.** *Soit N un réseau de Petri et $m \in \mathcal{M}_N$. Le marquage m est mort si et seulement si $\forall t \in T, \neg m[t\rangle$.*

L'ensemble d'accessibilité est le sous-ensemble (fini ou infini) des marquages du réseau qui sont accessibles à partir du marquage initial par le franchissement d'une séquence de transitions. Cet ensemble est aussi appelé *espace d'état*.

Définition 1.15 (Ensemble d'accessibilité)**.** *Soit (N, m_0) un réseau de Petri marqué. L'ensemble d'accessibilité de (N, m_0), noté $\mathcal{R}(N, m_0)$, est le plus petit ensemble tel que $m_0 \in \mathcal{R}(N, m_0)$, et si $\exists m \in \mathcal{R}(N, m_0), t \in T \mid m[t\rangle m'$ alors $m' \in \mathcal{R}(N, m_0)$.*

Le graphe d'accessibilité est un graphe dirigé (fini ou infini) qui décrit complètement la sémantique du réseau. Ses nœuds sont les marquages accessibles du réseau. Les arcs expriment les transitions possibles entre ces marquages.

Définition 1.16 (Graphe d'accessibilité)**.** *Soit (N, m_0) un réseau de Petri marqué. Le graphe d'accessibilité de (N, m_0) est un tuple $\langle V, E, I\rangle$ où*
– $V = \mathcal{R}(N, m_0)$ est l'ensemble des nœuds
– $E = \{(m, m') \in V \times V \mid \exists t \in T \mid m[t\rangle m'\}$ est l'ensemble des arcs
– $I = \{m_0\}$ est l'ensemble des nœuds initiaux

Le langage d'un réseau désigne l'ensemble des séquences qu'il permet.

Définition 1.17 (Langage d'un réseau)**.** *Soit (N, m_0) un réseau de Petri.*
– $\mathscr{L}(N, m_0) = \{\sigma \in T^* \mid m_0[\sigma\rangle\}$ *est le langage des séquences finies.*
– $\mathscr{L}^\infty(N, m_0) = \{\sigma \in T^\infty \mid m_0[\sigma\rangle\}$ *est le langage des séquences infinies.*
– $\mathscr{L}^{\max}(N, m_0) = \{\sigma \in T^* \mid m_0[\sigma\rangle m \wedge \forall t \in T, \neg m[t\rangle\}$ *est le langage des séquences maximales.*

La définition suivante introduit quelques propriétés usuelles des réseaux de Petri.

Définition 1.18 (Propriétés usuelles)**.** *Soit (N, m_0) un réseau de Petri marqué.*
– (N, m_0) *est* **pseudo-vivant** *si* $\forall m \in \mathcal{R}(N, m_0), \exists t \in T$ *tel que* $m[t\rangle$
– (N, m_0) *est* **quasi-vivant** *si :* $\forall t \in T, \exists m \in \mathcal{R}(N, m_0)$ *tel que* $m[t\rangle$
– (N, m_0) *est* **vivant** *si :* $\forall m \in \mathcal{R}(N, m_0), t \in T, \exists \sigma \in T^*$ *tel que* $m[\sigma.t\rangle$
– (N, m_0) *a un* **état d'accueil** $m_a \in \mathcal{M}_N$ *si :* $\forall m \in \mathcal{R}(N, m_0), \exists \sigma \in T^*$ *tel que* $m[\sigma\rangle m_a$
– (N, m_0) *admet une* **séquence infinie** *si :* $\exists \sigma \in T^\infty$ *tel que* $m_0[\sigma\rangle$
– (N, m_0) *est* **borné** *si :* $\exists k \in \mathbb{N}$ *tel que* $\forall m \in \mathcal{R}(N, m_0), p \in P, m(p) \leq k$
$p \in P$ *est k-bornée si :* $\exists k \in \mathbb{N}$ *tel que* $\forall m \in \mathcal{R}(N, m_0), m(p) \leq k$

Nous dirons qu'une place p est saine si elle est 1-bornée. Par extension, un réseau est sain si toutes ses places sont 1-bornées.

Les flots sont utilisés pour caractériser les propriétés invariantes du réseau. De nombreuses méthodes d'analyse structurelle se basent sur l'existence de flots particuliers. Le vecteur $\overrightarrow{0}$ est le vecteur nul dont la dimension est déterminée par le contexte.

Définition 1.19 (Flots)**.** *Soit N un réseau de Petri.*
– *Un P-flot f est un vecteur non nul de $\mathbb{Z}^P$ qui vérifie ${}^t f \cdot W = \overrightarrow{0}$.*
– *Un P-semiflot f est un vecteur non nul de $\mathbb{N}^P$ qui vérifie ${}^t f \cdot W = \overrightarrow{0}$.*
– *Un T-flot f est un vecteur non nul de $\mathbb{Z}^T$ qui vérifie $W \cdot f = \overrightarrow{0}$.*
– *Un T-semiflot f est un vecteur non nul de $\mathbb{N}^T$ qui vérifie $W \cdot f = \overrightarrow{0}$.*
Soit (N, m_0) un réseau de Petri marqué. f est un P-semiflot binaire si et seulement si il vérifie $\sum_{p \in P} f(p) \cdot m_0(p) = 1$. Le support d'un flot f, noté $||f||$ est l'ensemble des places couvertes par le flot : $||f|| = \{p \in P \mid f(p) \neq 0\}$.

Le terme de *flot positif* est souvent employé pour caractériser un semiflot.

1.3 Les réseaux de Petri colorés

Cette section définit les réseaux de Petri colorés et présente la classe qui sera étudiée dans cette thèse. Afin de lever toute ambiguïté nous qualifierons de réseaux de Petri ordinaires, ou réseaux ordinaires, les réseaux de Petri définis dans la section précédente.

1.3.1 Définition formelle des réseaux de Petri colorés

Les réseaux de Petri colorés [7, 8], ou réseaux colorés, offrent une possibilité d'expression plus compacte que les réseaux de Petri ordinaires. Dans un réseau coloré une place contient des jetons typés et une transition peut être franchie de différentes manières. Dans la terminologie des réseaux de Petri colorés, les types associés aux places et aux transitions sont appelés des domaines de couleur et un élément d'un domaine de couleur est appelé une couleur.
Un marquage d'un réseau de Petri coloré associe a toute place du réseau un multi-ensemble de jetons typés par le domaine de couleur de la place correspondante.
En général, une association place-couleur (ou transition-couleur) est appelée une instance de place (ou de transition).
Un arc joignant une place et une transition est étiqueté par une application linéaire appelée fonction de couleur. Les fonctions de couleur déterminent les jetons typés retirés ou ajoutés à une place

pour le franchissement d'une transition pour une couleur donnée, i.e., pour le franchissement d'une instance de transition.
Enfin les transitions du réseau peuvent être gardées. Une garde est une condition booléenne sur la couleur de la transition qui limite la franchissabilité de la transition aux couleurs pour lesquelles la garde est vraie.

Définition 1.20 (Réseau de Petri coloré)**.** *Un réseau de Petri coloré est un sextuplet* $\langle P, T, C, W^-, W^+, \phi\rangle$ *où* P *est un ensemble fini de* **places** *;* T *est un ensemble fini de* **transitions** *tel que* $P \cap T = \emptyset$ *;* C*, la fonction de couleur de* $P \cup T$ *vers* Σ *un ensemble fini d'ensembles finis et non vides ;* W^- *(resp.* W^+*) est une* **fonction d'incidence arrière (avant)***, qui associe à chaque couple* $(p,t) \in P \times T$ *une fonction de couleur de* $C(t)$ *vers* $Bag(C(p))$ *;* ϕ*,* **la fonction de garde***, associe à toute transition* t *une fonction de* $C(t)$ *vers* $\mathbb{B}$*. La* **fonction d'incidence** W *est définie par* $W(p,t) = W^+(p,t) - W^-(p,t), \forall (p,t) \in P \times T$*.*

La représentation graphique d'un réseau coloré suit les mêmes conventions que la représentation graphique d'un réseau ordinaire, à ceci près que les étiquettes des arcs ne sont plus des entiers mais des fonctions de couleur. Les gardes des transitions sont généralement placées à coté des transitions et écrites entre crochets. Les domaines de couleur des places et transitions seront parfois omis dans la mesure où ils peuvent généralement être aisément déduits des fonctions de couleur du réseau.
De plus si un réseau coloré N, N' ou N_u est défini le tuple qui le définit $\langle P, T, C, W^-, W^+, \phi\rangle$, $\langle P', T', C', W^{-\prime}, W^{+\prime}, \phi'\rangle$, ou $\langle P_u, T_u, C_u, W^-{}_u, W^+{}_u, \phi_u\rangle$ est lui aussi implicitement défini.

Définition 1.21 (Marquage coloré)**.** *Soit* N *un réseau de Petri coloré. Un marquage de* N *est une fonction qui à toute place* $p \in P$ *associe un élément de* $Bag(C(p))$*. L'ensemble des marquages de* N *est noté* $\mathcal{M}_N$*.*

Définition 1.22 (Règle de tir coloré)**.** *Soit* N *un réseau de Petri coloré et* $m \in \mathcal{M}_N$*. La transition* $t \in T$ *est franchissable pour l'instance* $c \in C(t)$ *en* m *si et seulement si* $\phi(t)(c) \wedge \forall p \in P, m(p) \geq W^-(p,t)(c)$*. Le franchissement de* (t,c) *en* m *amène à un marquage* m' *défini par* $\forall p \in P, m'(p) = m(p) + W(p,t)(c)$*. Nous noterons :*
- $m[(t,c)\rangle$ *la franchissabilité de l'instance* (t,c) *en* m
- $m[(t,c)\rangle m'$ *(ou* $Succ(m,(t,c)) = m'$*) si le franchissement de l'instance* (t,c) *en* m *mène au marquage* m'

La notion de franchissabilité ou de franchissement d'une séquence se déduit trivialement de la définition précédente et de sa version ordinaire. De même la définition de l'espace d'état et du graphe d'accessibilité d'un réseau coloré découle totalement de la définition précédente.

Un réseau coloré n'est qu'une abréviation d'un réseau de Petri ordinaire. L'opération qui consiste à obtenir le réseau ordinaire équivalent est appelé *dépliage*. Le principe du dépliage est simple. Pour chaque instance de place dans le réseau coloré il y a une place dans le réseau déplié. De même, pour chaque instance de transition dans le réseau coloré, pour laquelle la garde est vraie, il y a une transition dans le réseau déplié. L'application des fonctions de couleur nous permet ensuite de déduire aisément les matrices d'incidence du réseau déplié. L'hypothèse de finitude des domaines de couleur est donc indispensable pour ce dépliage.

Définition 1.23 (Réseau déplié)**.** *Soit* N *un réseau de Petri coloré. Le réseau déplié de* N *est un réseau de Petri* N_d *tel que :*
- $P_d = \{(p,c) \mid p \in P, c \in C(p)\}$
- $T_d = \{(t,c) \mid t \in T, c \in C(t), \phi(t)(c)\}$
- $\forall (p,c_p) \in P_d, (t,c_t) \in T_d, W_d^-((p,c_p),(t,c_t)) = W^-(p,t)(c_t)(c_p)$
- $\forall (p,c_p) \in P_d, (t,c_t) \in T_d, W_d^+((p,c_p),(t,c_t)) = W^+(p,t)(c_t)(c_p)$

Soit $m \in \mathcal{M}_N$*. Le marquage déplié de* m *est le marquage* $m_d \in \mathcal{M}_{N_d}$ *défini par :* $\forall (p,c_p) \in P_d, m_d((p,c_p)) = m(p)(c_p)$*. Le réseau déplié marqué de* (N,m) *est* (N_d, m_d)*.*

En pratique, le dépliage n'est pas toujours possible en raison de la taille des domaines de couleur du réseau. De plus, comme l'a remarqué Haddad dans [5] :

> "Tout l'intérêt d'une théorie des réseaux colorés est d'étudier les réseaux colorés sans avoir recours au dépliage du réseau."

Toute utilisation des réseaux colorés qui passe par un dépliage du réseau est en effet inutile dans la mesure ou les réseaux colorés ne sont alors utilisés que comme une facilité, ou un raccourci, syntaxique et non comme un outil mathématique.

Pour clore cette introduction formelle des réseaux colorés, nous préciserons que de nombreuses notations qui ont été définies pour les réseaux ordinaires sont aussi valables pour les réseaux colorés : $\mathcal{R}(N, m_0)$, et ${}^\bullet e, e^\bullet$ avec e une place ou une transition d'un réseau coloré,... Enfin nous noterons $\vec{\emptyset}$ la fonction de couleur qui à tout élément associe le multi-ensemble vide. Le domaine et le co-domaine de cette fonction seront définis par le contexte.

1.3.2 Propriété des fonctions de couleur

Certaines propriétés des fonctions de couleur des réseaux sont parfois utilisées pour caractériser la structure du réseau déplié. La définition suivante introduit quelques unes de ces propriétés. Rappelons que si f est une fonction de C vers $Bag(C')$, $f(c)$ est un multi-ensemble sur C'. Par conséquent, $f(c)(c')$ est la multiplicité de l'élément c' dans l'image de c par f.

Définition 1.24. *Soient C, C' deux ensembles et f une fonction de C vers $Bag(C')$. La fonction f est*
- **unitaire** *si et seulement si :*
  $\forall c \in C, c' \in C', f(c)(c') \leq 1$
- **jeton-unitaire** *si et seulement si :*
  $\forall c \in C, c'_1, c'_2 \in C',\ f(c)(c'_1) > 0 \wedge f(c)(c'_2) > 0 \Rightarrow c'_1 = c'_2$
- **quasi-injective** *si et seulement si :*
  $\forall c' \in C', c_1 \in C, c_2 \in C, f(c_1)(c') > 0 \wedge f(c_2)(c') > 0 \Rightarrow c_1 = c_2$.
- **quasi-surjective** *si et seulement si :*
  $\forall c' \in C', \exists c \in C \mid f(c)(c') > 0$.
- **quasi-totale** *si et seulement si :*
  $\forall c \in C, \exists c' \in C' \mid f(c)(c') > 0$.
- **orthonormale** *si et seulement : f est unitaire et*
  $\forall c \in C, \exists! c' \in C' \mid f(c)(c') = 1$ *et* $\forall c' \in C', \exists! c \in C \mid f(c)(c') = 1$
- **ortho-projection** *si et seulement si : f est unitaire et*
  $\forall c \in C, \exists! c' \in C' \mid f(c)(c') = 1$

Soient t une transition et p une place reliées par un arc étiquetée par une fonction de couleur f. Ces différentes propriétés induisent la structure suivante dans la réseau déplié.
f est **unitaire** $\Rightarrow$ La valuation d'un arc entre une place (p, c_p) et une transition (t, c_t) est 1.
f est **jeton-unitaire** $\Rightarrow$ Une transition (t, c_t) ne peut pas être liée à plusieurs (p, c_p).
f est **quasi-injective** $\Rightarrow$ Toute place (p, c_p) est liée à au plus une transition (t, c_t).
f est **quasi-surjective** $\Rightarrow$ Toute place (p, c_p) est liée à au moins une transition (t, c_t).
f est **quasi-totale** $\Rightarrow$ Toute transition (t, c_t) est liée à au moins une place (p, c_p).
f est **orthonormale** $\Rightarrow$ Toute place (p, c_p) est liée à exactement une transition (t, c_t), et inversement toute transition (t, c_t) et liée à exactement une place (p, c_p). De plus, la valuation de tous les arcs qui lient ces places et transitions est de 1.
f est une **ortho-projection** $\Rightarrow$ Toute transition (t, c_t) et liée à exactement une place (p, c_p). De plus, la valuation de tous les arcs qui lient ces places et transitions est de 1.

1.3.3 Manipulation des fonctions de couleur

Une difficulté majeure lorsque l'on travaille avec des réseaux colorés est que la structure du réseau coloré et de son déplié coïncident rarement. Il suffit en effet de reprendre la définition du dépliage pour constater que la matrice d'incidence du réseau déplié est obtenu par une application

des fonctions de couleur. C'est pourquoi l'analyse des réseaux colorés passe fréquemment par une manipulation de ces fonctions. Les opérateurs de transposition et de composition et les filtres de fonctions sont généralement utilisés à cette fin pour définir des relations structurelles entre des nœuds du réseau. Ces relations structurelles permettent en effet de caractériser la structure du réseau déplié de façon symbolique, i.e., sans procéder au dépliage.

Tout d'abord, nous ne sommes généralement pas intéressés par le montant exact de jetons produits par une fonction de couleur, mais plutôt par le fait que des jetons sont effectivement produits. On utilise pour cela l'opérateur $\overline{f}$ qui permet de passer d'une fonction de C vers $Bag(C')$ à une fonction de C vers $\mathcal{P}(C')$.

Définition 1.25. *Soient C, C' deux ensembles et f une fonction de C vers $Bag(C')$. La fonction $\overline{f}$ de C vers $\mathcal{P}(C')$ est définie par : $\forall c \in C, \overline{f}(c) = \{c' \in C' | f(c)(c') > 0\}$.*

Afin de pouvoir introduire les définitions de la transposition et de la composition, il est nécessaire d'étendre les fonctions de C vers $Bag(C')$ à des fonctions de $Bag(C)$ vers $Bag(C')$ à l'aide des deux règles suivantes :
- $f(\lambda.c) = \lambda.f(c)$
- $f(c_1 + c_2) = f(c_1) + f(c_2)$

Les fonctions de C vers $\mathcal{P}(C')$ sont aussi étendues à des fonctions de $\mathcal{P}(C)$ vers $\mathcal{P}(C')$ à l'aide des deux règles suivantes :
- $f(\emptyset) = \emptyset$
- $f(c_1 \cup c_2) = f(c_1) \cup f(c_2)$

La transposition est généralement utilisée pour trouver les instances d'une transition liées à une instance de place.

Définition 1.26 (Transposition)**.** *Soient C, C' deux ensembles et f une fonction de $Bag(C)$ vers $Bag(C')$. La fonction ${}^t f$ de $Bag(C')$ vers $Bag(C)$ est définie par : $\forall c \in C, c' \in C', {}^t f(c')(c) = f(c)(c')$.*

La composition est utilisée pour trouver les instances d'une transition (respectivement d'une place) liées à une autre instance de transition (respectivement de place) par une place (respectivement transition) intermédiaire.

Définition 1.27 (Composition)**.** *Soient C, C', C'' trois ensembles, f une fonction de $Bag(C'')$ vers $Bag(C')$ et g une fonction de $Bag(C)$ vers $Bag(C'')$. La fonction $f \circ g$ de $Bag(C)$ vers $Bag(C')$ est définie par : $\forall c \in C, c' \in C', (f \circ g)(c)(c') = \sum_{c'' \in C''} f(c'')(c') \cdot g(c)(c'')$.*

Les filtres quant à eux permettent de restreindre la production d'éléments sous certaines conditions. Il existe deux types de filtre :
- le pré-filtre est une condition sur le domaine de la fonction
- le post-filtre est une condition sur le co-domaine de la fonction

Définition 1.28 (Fonction filtrée)**.** *Soient C, C' deux ensembles, f une fonction de $Bag(C)$ vers $Bag(C')$, ψ une fonction de C vers $\mathbb{B}$ et ψ' une fonction de C' vers $\mathbb{B}$. La fonction $[\psi]f[\psi']$ de $Bag(C)$ vers $Bag(C')$ est définie par : $\forall c \in C, c' \in C', [\psi]f[\psi'](c)(c') = f(c)(c')$ si $\psi(c)$ et $\psi'(c')$, 0 sinon. ψ est appelée le pré-filtre et ψ' est appelée le post-filtre de la fonction f.*

Nous pouvons faire les remarques suivantes :
- Toute fonction sera considérée comme une fonction filtrée. Par défaut, ses pré- et post-filtres seront la constante *true*.
- La définition des filtres et des opérateurs de transposition et de composition se déduit trivialement pour des fonctions allant d'un ensemble $\mathcal{P}(C)$ vers un autre ensemble $\mathcal{P}(C')$. Par exemple, si f est une fonction de $\mathcal{P}(C)$ vers $\mathcal{P}(C')$, ${}^t f$ est la fonction de $\mathcal{P}(C')$ vers $\mathcal{P}(C)$ définie par ${}^t f(c') = \{c | c' \in f(c)\}$

Enfin, il peut être utile de vérifier que l'image d'une fonction de $\mathcal{P}(C)$ vers $\mathcal{P}(C')$ est incluse dans l'image d'une autre fonction de $\mathcal{P}(C)$ vers $\mathcal{P}(C')$.

Définition 1.29. *Soit f et g deux fonctions de $\mathcal{P}(C)$ vers $\mathcal{P}(C')$. Si $\forall c \in C, f(c) \subseteq g(c)$ alors nous notons $f \sqsubseteq g$.*

La figure 1.1 permet d'illustrer une utilisation typique de ces relations structurelles. Considérons une instance (t, c_t). Nous souhaitons calculer les instances de u qui déposent dans p des jetons nécessaire au tir de (t, c_t). Notre première étape sera de trouver ces jetons. Par définition, ils appartiennent à l'ensemble $\overline{\Psi}(c_t) = \{c_p \mid \Psi(c_t)(c_p) > 0\}$. Nous devons maintenant calculer les instances de h qui produisent ces jetons. L'ensemble des instances de u qui déposent des jetons de type c_p dans une place p est l'ensemble $\overline{{}^t\Gamma}[\Phi](c_p)$. Nous appliquons en effet à cette fonction le post-filtre Φ afin de retirer de cet ensemble les instances de u pour lesquelles la garde n'est pas évaluée à *true*. Finalement, pour trouver les instances de u qui déposent dans p des jetons nécessaires au franchissement de (t, c_t), il nous suffit de composer ces deux fonctions. L'ensemble recherché est donc :

$$\overline{{}^t\Gamma}[\Phi] \circ \overline{\Psi}(c_t)$$

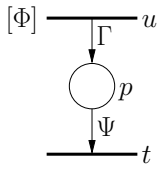


FIG. 1.1 – Manipulation des fonctions de couleur

1.3.4 Les flots colorés

Les définitions énoncées précédemment nous permettent de définir la notion de flot coloré. Cette définition est tirée de [2]. Un flot est une somme pondérée de places associées à un domaine de couleur : le domaine de couleur de l'invariant, qui peut être considéré comme étant son domaine d'interprétation. Chaque poids associé aux places est une fonction ayant pour domaine le domaine de la place et pour co-domaine le domaine de l'invariant.
Dans la définition suivante, $Bag_{\mathbb{Q}}(A)$ désigne le $\mathbb{Q}$-espace vectoriel canonique sur l'ensemble non vide A. C'est l'ensemble des fonctions de A vers $\mathbb{Q}$.

Définition 1.30 (Flots colorés)**.** *Soit N un réseau de Petri coloré. Un flot coloré f de domaine C_f est une somme formelle $\sum_{p \in P} f(p)$ avec $\forall p \in P$, $f(p)$ est une application de $Bag_{\mathbb{Q}}(C(p))$ vers $Bag_{\mathbb{Q}}(C_f)$ et telle que $\forall t \in T, \sum_{p \in P} f(p) \circ W(p,t) = \vec{\emptyset}$. Si $\forall p \in P$, $f(p)$ est une application de $Bag(C(p))$ vers $Bag(C_f)$ alors f est un flot coloré positif ou semiflot coloré.*

Nous pouvons illustrer cette définition à l'aide du réseau de la figure 1.2. Ce réseau coloré respecte une certaine "syntaxe" qui sera formellement définie dans la suite de ce chapitre. Il modélise une solution au problème des philosophes dans laquelle un philosophe prend ses deux fourchettes atomiquement.
Les places *Idle* et *Eating* modélisent les différent états d'un philosophe p : inactif ou en train de manger. La place *Forks* modélise l'état des fourchettes disposées sur la table. Un jeton p dans cette place indique que la fourchette p est libre. Enfin, les transitions *Seat* et *Leave* modélisent la prise des fourchettes par un philosophe et la levée de table (avec pose des fourchettes). Le franchissement de la transition *Seat* pour la valeur p retire un jeton p de la place *Idle* et les jetons p et *succ* p de

la place $Forks$ puis dépose un jeton p dans la place $Eating$. La fonction $succ$ de E dans E permet de sélectionner le successeur d'un élément de l'ensemble ordonné E[1].

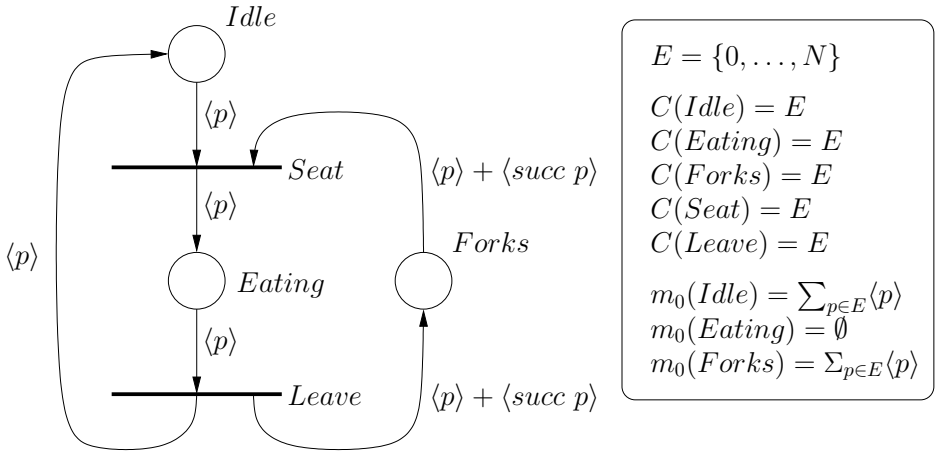


FIG. 1.2 – Une solution au problème des philosophes

Pour ce réseau, $\mathcal{F}_1$, de domaine P, et défini par

$$\mathcal{F}_1 = \langle p \rangle \cdot Idle + \langle p \rangle \cdot Eating$$

est un invariant. Il s'interprète comme :

$$\forall p \in E, Idle(p) + Eating(p) = 1$$

Il peut aussi être interprété par l'assertion suivante : "un philosophe est soit inactif, soit à table".

Un autre invariant, $\mathcal{F}_2$, de domaine E garantit qu'une fourchette p est libre ou prise par un des deux philosophes qui peuvent la réclamer : les philosophe p et $pred\ p$ ($pred$ étant définie comme la fonction inverse de $succ$). Il est défini par :

$$\mathcal{F}_2 = \langle p \rangle \cdot Forks + (\langle p \rangle + \langle pred\ p \rangle) \cdot Eating$$

Le réseau de la figure 1.3 représente le dépliage du réseau coloré 1.2 pour une valeur de N égale à 2. Pour faciliter la lisibilité, nous avons dupliqué la place $Fork0$ (que nous avons alors mis en pointillé).

On observe ici que le réseau est nettement moins lisible que le réseau initial coloré même pour une faible valeur de N. Le comportement du premier philosophe est modélisé par les places $Idle0$, $Fork0$ et$Eating0$ et les transitions $Seat0$ et $Leave0$.

On peut alors y transposer les 2 invariants identifiés précédemment. L'invariant $\mathcal{F}_1$ qui stipule que soit la place $IdleX$, soit la place $EatingX$ sont marquées est assez aisé à voir. Pour le second invariant $\mathcal{F}_2$, prenons par exemple le cas de la fourchette du philosophe 1 : le jeton initialement placé dans la place $Fork1$ peut se trouver soit dans $Fork1$ si la fourchette n'est utilisée par aucun philosophe, soit dans $Eating1$ si le philosophe 1 décide de manger, soit dans la place $Eating0$ si le philosophe 0 décide de manger. Dans ce dernier cas, le philosophe 0 récupère sa fourchette (place $Fork0$) et celle de son voisin (place $Fork1$).

[1] le successeur de N par la fonction $succ$ est 0.

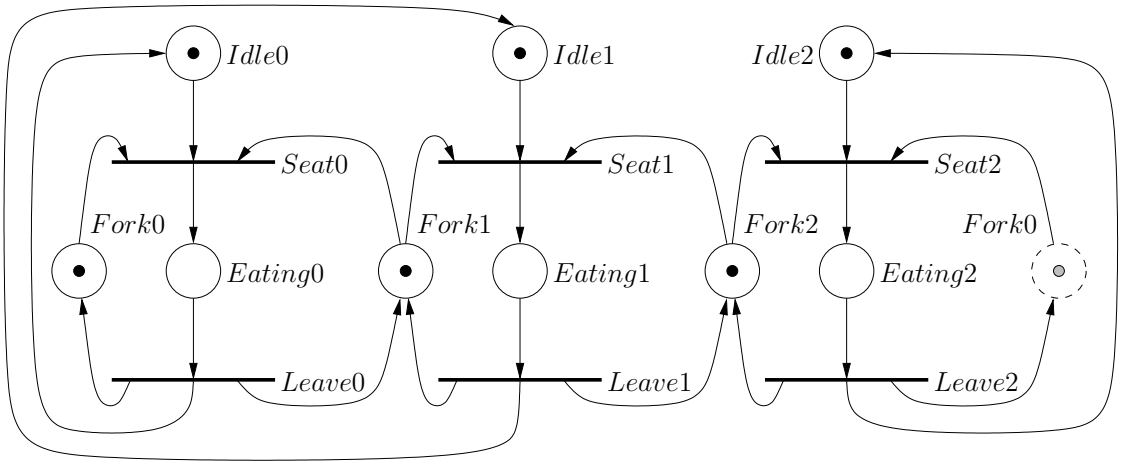


FIG. 1.3 – Le réseau de la figure 1.2 déplié (pour $N = 2$).

Les invariants colorés peuvent alors être traduits comme suit pour le réseau déplié :
- pour l'invariant $\mathcal{F}_1$:
  - $Idle0 + Eating0 = 1$
  - $Idle1 + Eating1 = 1$
  - $Idle2 + Eating2 = 1$
- pour l'invariant $\mathcal{F}_2$:
  - $Eating0 + Fork0 + Eating1 = 1$
  - $Eating1 + Fork1 + Eating2 = 1$
  - $Eating2 + Fork2 + Eating0 = 1$

Comme nous le voyons ici, les réseaux colorés nous permettent de faciliter la modélisation de systèmes complexes comparé aux réseaux de Petri ordinaires. Nous allons maintentant présenter une classe particulière de réseaux de Petri colorés qui permet l'utilisation de jeton contenant des structures de données de haut-niveau. Ceci permet alors de modéliser plus facilement des systèmes concurrents tout en conservant tout ou partie des capacités d'analyse des réseaux de Petri ordinaires. Il est alors intéressant de pouvoir raisonner sur le modèle coloré sans avoir à déplier le réseau afin de faciliter l'analyse du système puisque les invariants sont paramétrés. Nous verrons cependant dans le chapitre 8 que le calcul de flots reste un problème complexe lorsque l'on passe au réseaux colorés.

1.3.5 Une classe de réseaux de Petri colorés

Les réseaux de Petri colorés généraux qui ont précédemment été définis dans cette section n'imposent aucune contrainte sur les domaines de couleur, les fonctions de couleur des matrices d'incidence, et les gardes des transitions du réseau. Une utilisation pratique de ces réseaux nécessite naturellement de fixer une syntaxe particulière pour ces éléments.
Plusieurs classes de réseaux colorés ont déjà été introduites dans la littérature. Nous pourrons citer les réseaux de Petri réguliers [5], et les réseaux de Petri bien formés [1] qui ont été introduits afin d'exploiter efficacement les symétries du modèle en vue de la construction du graphe des marquages symboliques. Ces classes sont bien adaptées pour la représentation de mécanismes de synchronisation complexes. Cependant, elles ne permettent pas de représenter efficacement les données, et ce pour deux raisons. D'une part, les types, ou classes de couleur dans la terminologie des réseaux de Petri bien formés, sont limités à des types énumérés et numériques. Il devient alors difficile de modéliser des types de données de haut niveau utilisés dans les langages de programmation tels que les types structurés ou les tableaux. D'autre part la syntaxe des fonctions de couleurs ne permet

pas d'effectuer des opérations complexes sur les données, tels que des opérations numériques.
La définition d'une classe de réseaux colorés résulte d'un compromis entre facilité d'expression et possibilités d'analyse. Une classe de réseaux restreinte permettra la définition d'algorithmes et de techniques exploitant ces restrictions, p.ex., les réseaux de Petri bien formés et la détection des symétries, alors qu'un réseau coloré appartenant à une classe n'imposant aucune contrainte sera difficilement analysable.
Nous proposons dans cette section une classe de réseaux colorés, adaptée pour la vérification de programmes concurrents. Notre classe possède des similitudes avec les réseaux de Petri bien formés [1], mais elle pallie les lacunes de celle-ci à représenter des données de "haut-niveau". En effet, les expressions qui apparaissent dans les fonctions de couleur ne respectent aucune syntaxe, contrairement aux fonctions prédéfinies du formalisme des réseaux de Petri bien formés (identité, successeur, diffusion,...). Nous verrons à l'aide d'un exemple que cette liberté offre une grande souplesse aux concepteurs de modèles.

Dans la suite de cette thèse, nous considérerons que tout réseau coloré satisfait les contraintes syntaxiques décrites plus bas. En particulier, lorsque nous parlerons de fonction de couleur, nous considérerons une fonction respectant cette syntaxe.

a. Les domaines de couleur

Tout d'abord, nous supposerons qu'il existe un ensemble de types de base qui pourront servir à la construction des domaines de couleur.

Définition 1.31 (Types de base)**.** *L'ensemble des types de base Δ est un ensemble fini d'ensembles finis et non vides.*

Un domaine de couleur est un produit cartésien de types de base.

Définition 1.32 (Domaines de couleur)**.** *Un domaine de couleur C est un produit cartésien $C_1 \times \cdots \times C_n$ avec $C_i \in \Delta, \forall i \in [1..n]$. Si $n = 0$ nous notons $C = \epsilon$. L'ensemble des domaines de couleur est noté $\mathcal{C}$.*

Un élément d'un domaine de couleur C sera noté $\langle c_1, \ldots, c_n \rangle$ ou encore c si $n = 1$. Ainsi, si p est une place de domaine $\{a, b, c\} \times \{a, b, c\}$, $(p, \langle a, b \rangle)$ est l'instance de la place p dont le premier élément du domaine est a et le second est b. Pour les transitions nous supposerons l'existence d'une application injective associant à chaque élément de son domaine une variable. Ces variables sont utilisées afin d'alléger l'écriture des fonctions de couleur. Considérons par exemple une transition t de domaine $\{a, b, c\} \times \mathbb{B}$. Si les deux éléments de ce domaine sont respectivement associés aux variables X et Y, $(t, \langle X = c, Y = true \rangle)$ et $(t, \langle c, true \rangle)$ désigneront la même instance.

b. Les fonctions de couleur

Les fonctions de couleur sont construites à partir d'expressions élémentaires qui sont des fonctions d'un domaine de couleur C vers un type de base δ. Il existe deux types d'expressions élémentaires : les variables et les expressions fonctionnelles.
Les variables (ou projections) permettent de sélectionner un élément du domaine de couleur C.
Les expressions fonctionnelles permettent quant à elles de réaliser des opérations complexes sur les types de base. Cette appellation regroupe l'ensemble des expressions correctes sur un type de base δ. Nous pouvons donc considérer que les variables sont des expressions fonctionnelles particulières, de même que les constantes.

Définition 1.33 (Expression élémentaire)**.** *Soient $C = C_1 \times \cdots \times C_n \in \mathcal{C}$ un domaine de couleur et $\delta \in \Delta$ un type de base. L'ensemble des expressions élémentaires de C vers δ est l'ensemble*

$$Expr_{C \to \delta} = \mathcal{V}_{C \to \delta} \cup \mathcal{F}_{C \to \delta}$$

où $\mathcal{V}_{C\to\delta}$ et $\mathcal{F}_{C\to\delta}$ sont les ensembles des variables et expressions fonctionnelles définis plus bas.

les variables

Soit $i \in [1..n]$. La variable V_i^C est une fonction de C vers C_i définie par :

$$\forall c = \langle c_1, \ldots, c_n \rangle \in C, V_i^C(c) = c_i$$

L'ensemble $\mathcal{V}_{C\to\delta}$ des variables de C vers δ est défini par :

$$\mathcal{V}_{C\to\delta} = \{V_i^C \mid i \in [1..n] \wedge C_i = \delta\}$$

les expressions fonctionnelles

Soient $\delta, \delta_1, \ldots, \delta_m \in \Delta$ des types de base, f une fonction de $\delta_1 \times \cdots \times \delta_m$ vers δ, et $e_1 \in Expr_{C\to\delta_1}, \ldots, e_m \in Expr_{C\to\delta_m}$ un ensemble d'expressions élémentaires. L'expression fonctionnelle $(f, (e_1, \ldots, e_m))$ est une fonction de C vers δ définie par :

$$\forall c \in C, (f, (e_1, \ldots, e_m))(c) = f(e_1(c), \ldots, e_m(c))$$

L'ensemble $\mathcal{F}_{C\to\delta}$ des expressions fonctionnelles de C vers δ est défini par :

$$\mathcal{F}_{C\to\delta} = \{(f, (e_1, \ldots, e_m)) | f \in \delta_1 \times \cdots \times \delta_m \to \delta \wedge \forall i \in [1..m], e_i \in Expr_{C\to\delta_i}\}$$

Afin d'alléger l'écriture des fonctions de couleur, nous préférerons généralement à la notation formelle des projections l'utilisation des variables associées aux domaines des transitions. Ainsi, $V_2^{C(t)}$ pourra directement être noté Y si le deuxième élément du domaine de t est associé à cette variable.

Exemple 1.1. *Soit $\delta = \{0..9\}$ un type de base et $C = \delta \times \mathbb{B}$ un domaine de couleur. Considérons les trois expressions élémentaires suivantes.*
- *V_1^C est une projection de C vers δ*
- *3 est une expression fonctionnelle de C vers δ. On la note formellement comme le couple $(3, \langle\rangle\rangle)$.*
- *$estPaire(V_1^C)$ est également une expression fonctionnelle de C vers $\mathbb{B}$ notée formellement comme le couple $(estPaire, \langle V_1^C\rangle)$. La fonction $estPaire$ de δ vers $\mathbb{B}$ est définie par $estPaire(i) = true$ si $i \mod 2 = 0, false$ sinon.*

Nous pouvons appliquer ces expressions à l'élément $\langle 5, 4\rangle \in C$:
- $V_1^C(c) = 5$
- $(3, \langle\rangle\rangle)(c) = 3$
- $(estPaire, \langle V_1^C\rangle)(c) = false$

Les tuples d'expressions élémentaires permettent de construire les fonctions de couleur du réseau. Un tuple produit un unique jeton typé de multiplicité supérieure ou égale à 1. Un tuple est caractérisé par :
- un facteur qui indique la multiplicité du jeton produit par le tuple
- une garde qui indique sous quelles conditions le jeton est effectivement produit
- une liste d'expressions qui définit le jeton produit

La notion de garde est définie plus tard dans cette section (définition 1.36).

Définition 1.34 (Tuple d'expressions élémentaires)**.** *Soient $C = C_1 \times \cdots \times C_n$, $C' = C'_1 \times \cdots \times C'_m \in \mathcal{C}$. L'ensemble des tuples d'expressions élémentaires de C vers $Bag(C')$, $Tup_{C\to Bag(C')}$ est l'ensemble des triplets $\langle \gamma, \alpha, E\rangle$ tel que :*
- *$\gamma \in \mathcal{G}_C$ est la garde du tuple*
- *$\alpha \in \mathbb{N}^+$ est le facteur du tuple*
- *$E = \langle ex_1, \ldots, ex_m\rangle$, avec $\forall i \in [1..m], ex_i \in Expr_{C\to C'_i}$, est la liste d'expressions du tuple*

Tout tuple $tup = \langle \gamma, \alpha, \langle ex_1, \ldots, ex_m\rangle\rangle$ de $Tup_{C\to Bag(C')}$ est défini par :

$$\forall c \in C, tup(c) = \begin{cases} \alpha \cdot \langle ex_1(c), \ldots, ex_m(c)\rangle & si\ \gamma(c) \\ \emptyset & sinon \end{cases}$$

Un tuple $\langle \gamma, \alpha, \langle ex_1, \ldots, ex_n \rangle \rangle$ sera généralement noté $[\gamma]\ \alpha \cdot \langle ex_1, \ldots, ex_n \rangle$.

Exemple 1.2. *Soit $\delta = \{0..9\}$ un type de base et $C = \delta \times \delta$ et $C' = \delta \times \delta \times \delta$ deux domaines de couleur. Le tuple $[V_1 > V_2]\ 3 \cdot \langle V_1, V_2, (V_1 + V_2) \mod 10 \rangle$ est un tuple valide de C vers $Bag(C')$. Appliqué à un élément $\langle c_1, c_2 \rangle$ du domaine de couleur C, il produit trois jetons de type $\langle c_1, c_2, (c_1 + c_2) \mod 10 \rangle$ si $c_1 > c_2$. Dans le cas contraire, il produit le multi-ensemble vide.*

Enfin une fonction de couleur est une somme de tuples d'expressions d'élémentaires.

Définition 1.35 (Fonction de couleur)**.** *Soient $C, C' \in \mathcal{C}$. Une fonction de couleur f de C vers $Bag(C')$ est une somme $f = \sum_{i \in [1..k]} tup_i$ telle que $\forall i \in [1..k], tup_i \in Tup_{C \to Bag(C')}$. La fonction f est définie par :*

$$\forall c \in C, f(c) = \sum_{i \in [1..k]} tup_i(c)$$

L'ensemble des fonctions de couleur de C vers $Bag(C')$ est noté $Map_{C \to Bag(C')}$.

c. Les gardes

Dans la classe de réseaux colorés définie les gardes peuvent être associées aux transitions du réseau et aux tuples qui apparaissent dans les fonctions de couleurs. Nous n'imposons aucune syntaxe particulière pour ces gardes. Toute expression booléenne peut donc être utilisée.

Définition 1.36 (Garde)**.** *Soit $C \in \mathcal{C}$. $\mathcal{G}_C$, l'ensemble des gardes sur C est l'ensemble $\mathcal{F}_{C \to \mathbb{B}}$.*

1.3.6 Un exemple : spécification d'un répartiteur de charges

Nous souhaitons spécifier un système de répartition de charges afin d'illustrer les possibilités offerte par la classe de réseaux colorés précédemment définie.

a. Spécification informelle du système

Le système modélisé fait interagir divers processus : les clients, les serveurs, et le répartiteur de charges.
Tous les clients du système ont un comportement identique et se contentent de répéter inlassablement la séquence d'opérations suivante :

1. envoyer une requête aux serveurs
2. attendre la réponse

Un serveur est initialement en attente d'une requête d'un client qui lui est adressée par l'intermédiaire du répartiteur de charge. Si il reçoit une telle requête il la traite en effectuant les calculs nécessaires, répond au client, puis se remet en attente d'une autre requête.
Le répartiteur de charges joue un rôle d'intermédiaire entre les clients et les serveurs du système. Il a deux fonctions :
Une fonction d'**aiguillage** : lorsqu'il reçoit une requête d'un client, il doit l'aiguiller vers le serveur adéquat.
Une fonction de **rééquilibrage** : lorsque la charge des serveurs évolue il doit procéder au rééquilibrage des charges.

b. Spécification formelle du système

Nous allons concevoir de façon modulaire le réseau coloré modélisant ce système en identifiant les types de données manipulés, puis en définissant les sous-réseaux correspondant aux différents acteurs.

i. Identification des types de base

Nous allons dans un premier temps définir les types de base nécessaires à la construction de notre réseau. Notons N le nombre de clients et M le nombre de serveurs. Les types C et S permettent d'identifier les clients et serveurs du système. Le répartiteur de charges doit mémoriser les requêtes envoyées aux serveurs. Nous définissons pour cela le type L qui est l'ensemble des fonctions de S vers $[0..N]$. Ainsi si $l \in L$ et $s \in S$, $l(s)$ est le nombre de requêtes adressées au serveur s.
La définition formelle de ces trois types est donnée plus bas.
$C = \{c_1, \ldots, c_N\}$
$S = \{s_1, \ldots, s_M\}$
$L = \{f \in S \to \{0..N\}\}$

ii. Les clients

Le réseau de la figure 1.4 modélise l'algorithme exécuté par les clients. Dans l'état initial du système, tous les clients sont dans l'état inactif (place $clientsIdle$ marquée par tous les éléments du type C). L'envoi d'une requête (transition $clientsSendRequest$) se traduit par le dépôt d'un jeton identifiant le client dans la place $clientsRequest$. Le client passe alors dans l'état d'attente de l'acquittement (place $clientsWaiting$). La réception de cet acquittement (réception $clientsReceiveAck$) par un client c est possible dès que le jeton $\langle c \rangle$ est présent dans la place $clientsAck$. Le client peut alors repasser dans l'état inactif.

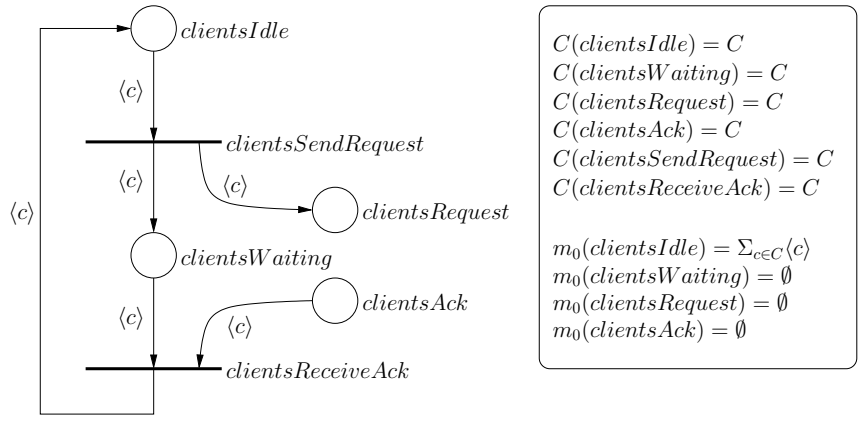
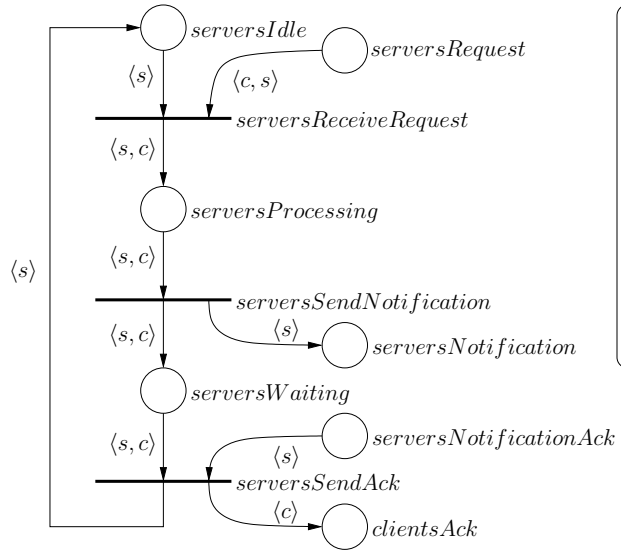


FIG. 1.4 – Algorithme des clients

iii. Les serveurs

L'algorithme exécuté par les serveurs est modélisé par le réseau de la figure 1.5. Tous les serveurs sont initialement dans l'état inactif (place $serversIdle$). La présence d'un jeton $\langle c, s \rangle$ dans la place $serversRequest$ indique que la requête du client c a été adressée au serveur s. Le tir de l'instance $(serversReceiveRequest, \langle c, s \rangle)$ correspond à la réception de cette requête. Le serveur passe alors dans l'état "traitement de la requête" (place $serversProcessing$). Notons que l'identifiant du client (c) est mémorisé pour que le serveur puisse l'acquitter plus tard. Une fois le calcul terminé, le serveur le signale au répartiteur (transition $serversSendNotification$) en déposant un jeton contenant son identifiant dans la place $serversNotification$. Cette notification est nécessaire : elle permet au répartiteur de mettre à jour ses informations sur les charges des serveurs et, le cas échéant, de procéder au rééquilibrage. Une fois l'acquittement de cette notification reçu (retrait par la transition $serversSendAck$ du jeton $\langle s \rangle$ de la place $serversNotificationAck$), le serveur peut acquitter le client en déposant le jeton $\langle c \rangle$ dans la place $clientAck$. Il repasse alors dans l'état inactif.



$C(serversIdle) = S$
$C(serversProcessing) = S \times C$
$C(serversWaiting) = S \times C$
$C(serversRequest) = C \times S$
$C(serversNotification) = S$
$C(serversNotificationAck) = S$
$C(serversReceiveRequest) = S \times C$
$C(serversSendNotification) = S \times C$
$C(serversSendAck) = S \times C$

$m_0(serversIdle) = \Sigma_{s \in S} \langle s \rangle$
$\forall p \neq serversIdle, m_0(p) = \emptyset$

FIG. 1.5 – Algorithme des serveurs

iv. Le répartiteur

La première fonction de routage des requêtes est modélisée par le réseau de la figure 1.6. La place *lbIdle* modélise l'état inactif du répartiteur. Son domaine est le type L précédemment défini. En associant un entier de 0 à N à chaque serveur, le répartiteur peut ainsi mémoriser le nombre de requêtes adressées à chaque serveur. L'élément *emptyLoad* initialement présent dans cette place est la fonction qui associe l'entier 0 à tout serveur : dans l'état initial aucun serveur n'a de requêtes à traiter. Le routage peut être effectué à l'aide d'une seule transition : la transition *lbRoute*. Ce routage consiste à prendre une requête envoyée par un client c, i.e., le jeton $\langle c \rangle$ présent dans la place *clientsRequest*, et à la rediriger vers le serveur le moins chargé, i.e., celui ayant le moins de requêtes à traiter. Afin d'identifier le serveur le moins chargé, nous introduisons la fonction *least* de L vers C définie comme suit :

$$least(l) = s \text{ tel que } \forall s' \in S, l(s') \geq l(s)$$

Une fois cette requête redirigée, le répartiteur met à jour ses informations en incrémentant la charge du serveur vers lequel la requête a été redirigée. Nous utilisons à cette fin la fonction *incr* de $L \times S$ vers S définie par :

$$incr(l, s) = l' \text{ avec } \forall s' \in S, l'(s') = \begin{cases} l(s') + 1 & \text{si } s' = s \\ l(s') & \text{sinon} \end{cases}$$

Notons l'apparition d'expressions fonctionnelles dans les fonctions de couleur de ce réseau : $least(l)$, $incr(l, least(l))$.
La dernière étape consiste à modéliser l'algorithme de rééquilibrage exécuté par le répartiteur de charges. Ce processus intervient lorsque le répartiteur reçoit de la part d'un serveur un message indiquant qu'il a traité une requête d'un client. Le jeton identifiant le serveur s est alors retiré de la place *serversNotification* par la transition *lbReceiveNotification* et la charge de ce serveur est décrémentée par le répartiteur. Nous utilisons pour cela la fonction *decr* de $L \times S$ vers L. Le répartiteur passe ensuite dans l'état "en cours de rééquilibrage" (place *lbBalancing*). Deux cas peuvent alors se présenter :
Les requêtes sont uniformément distribuées sur les serveurs (transition *lbNoBalance*).
Le répartiteur n'exécute aucun traitement et retourne dans l'état inactif.

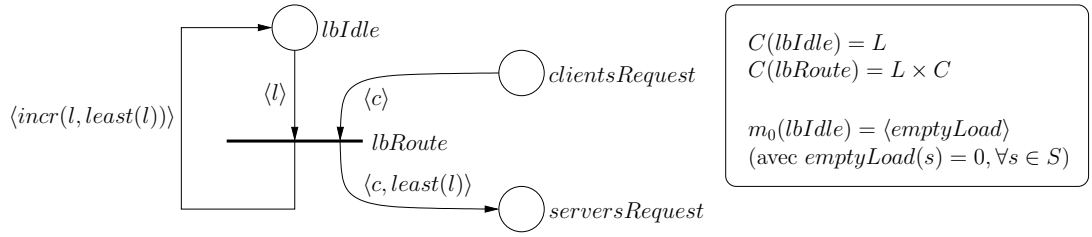


FIG. 1.6 – Algorithme de routage du répartiteur

Il y a un déséquilibre dans l'affectation des requêtes (transition *lbBalance*).

Le répartiteur doit alors procéder au rééquilibrage. Il retire une requête adressée au serveur le plus chargé, i.e., $most(l)$, pour la rediriger vers le serveur le moins chargé, i.e., $least(l)$. Il retourne ensuite dans l'état inactif en mettant à jour ses informations : il incrémente la charge du serveur vers lequel il a redirigé une requête et décrémente la charge du serveur pour lequel il a retiré une requête.

Pour tester si le rééquilibrage est nécessaire, nous utilisons la fonction *balanced* de L vers $\mathbb{B}$ définie par :

$$balanced(l) \Leftrightarrow \forall s, s' \in S, |l(s) - l(s')| \leq 1$$

Autrement dit, il ne peut pas y avoir deux serveurs s et s' pour lesquels la différence de charges est de 2 au plus (auquel cas une requête adressée à l'un devrait être redirigée vers l'autre).

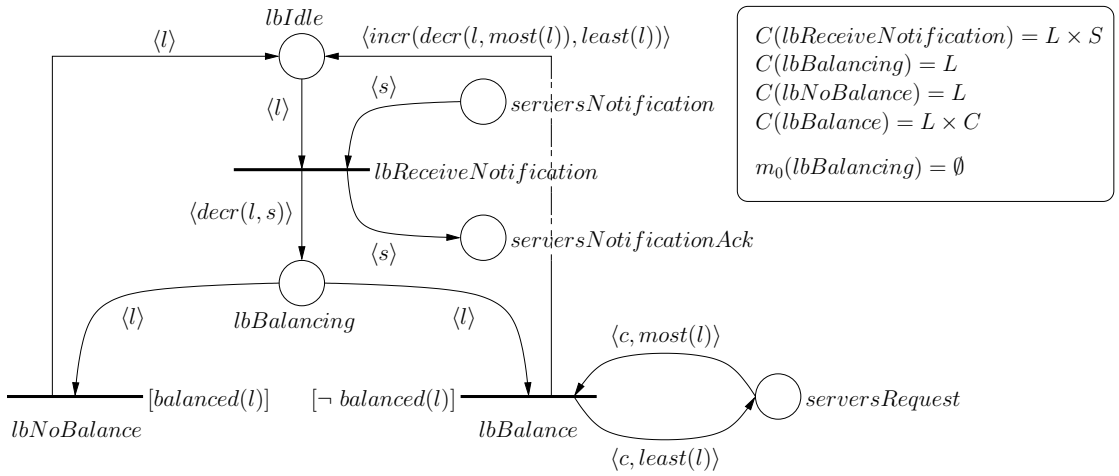


FIG. 1.7 – Algorithme de rééquilibrage du répartiteur

Nous avons pu, à l'aide de cet exemple, illustrer l'utilité de notre classe de réseaux. La possibilité d'insérer des expressions fonctionnelles dans les gardes et fonctions de couleur laisse aux concepteurs une grande souplesse. Cependant, nous n'avons pas pour autant sacrifié à cette facilité d'expression la possibilité de définir des techniques d'analyse efficaces.

Chapitre

2 Systèmes parallèles et répartis

Sommaire

Dans ce chapitre, nous présenterons brièvement quelques notion de systèmes systèmes répartis et multiprocesseurs. Dans un premier temps (section 2.1) nous nous attarderons sur les principales caractéristiques des systèmes à mémoire répartie. Nous y aborderons les notions de réseau, de modes de communications, de mécanismes de synchronisation et de détection de la terminaison. Nous effectuerons également un bref descriptif des principales caractéristiques des algorithmes dans ce type d'environnement.

Nous nous attacherons ensuite (section 2.2) à présenter les principales notions liées aux systèmes à mémoire partagée : notion de *thread* (ou processus léger), mécanisme de synchronisation ainsi que les problèmes couramment rencontrés dans ce type de systèmes.

Nous ne prétendons pas être exhaustifs mais présentons uniquement un rapide aperçu des principales caractéristiques de ces deux types de systèmes. Essentiellement, nous présenterons des concepts utilisés lors de cette thèse.

2.1 Systèmes répartis

La notion de systèmes à mémoire répartie est indissociable de celle de *cluster* (ou grappe) de stations de travail (*Network Of Workstations* – NOW). Il s'agit d'un ensemble de stations de travail interconnectées par un réseau local mettant leurs ressources en commun pour permettre à une système parallèle d'utiliser ces machines pour les faire travailler de façon coopérative dans un but commun.

2.1.1 Réseaux de machines

L'utilisation d'un réseau de machines peut permettre la résolution de problèmes complexes qui ne peuvent – ou peuvent difficilement – être résolus sur des machines simples tant pour des raisons de manque de puissance de calcul que de ressources mémoire limitées. C'est notamment ce dernier point que la parallélisation d'un programme séquentiel dans un environnement à mémoire réparti cherche à résoudre.

Le gain en mémoire est évident et garanti pour chaque programme parallélisé. Le gain de temps étant quant à lui fortement lié au degré de parallélisation du programme parallèle, un degré de parallélisation faible aboutira à une exécution semblable à une exécution séquentielle et ne débouchera donc pas sur une réduction du temps d'exécution. Plus le degré de parallélisation est élevé, plus le gain de temps sera important. La formule permettant d'obtenir le temps d'exécution théorique d'un programme parallélisé est donc :

$$T^N_{théorique} = \frac{T}{N}$$

où T est le temps d'exécution en séquentiel et N le nombre de machines utilisées pour la parallélisation. Ce gain reste cependant théorique puisqu'il ne tient pas compte de certains facteurs relatifs à la parallélisation qui vont plus ou moins limiter ce gain théorique : temps de communications, sérialisation/déserialisation des données échangées, algorithmique supplémentaire nécessaire à la parallélisation, ...

a. Topologies

Un réseau de machines est défini par un ensemble de critères parmi lesquels figurent le type de liaison utilisé, les protocoles utilisés dans les différentes couche du modèle OSI [141, 143] ou encore la topologie physique de réseau. Nous présentons quelques topologies pour les réseaux locaux.

i. Topologie point-à-point

La topologie point-à-point (fig. 2.1) est une topologie dans laquelle chaque machine est connectée directement avec toutes les autres machines du réseau. Chaque nœud du réseau possède donc N liens de communications où N représente le nombre de nœuds total. Cette topologie très efficace en terme de communications – puisque chaque message est transmis directement de la source à la destination en choisissant le lien correspondant – reste cependant très chère en terme d'équipement et difficilement réalisable dès que le nombre de machines devient important.

ii. Topologie en anneau

La topologie en anneau relie chaque machine à deux voisins sur le réseau. Moins gourmande en terme de liaisons de communications, cette topologie dégrade les performances en terme de communications ; si l'on considère N machines interconnectées l'envoi d'un message d'une machine à son opposée peut nécessiter jusqu'à $N/2$ retransmissions. Elle est également fortement vulnérable

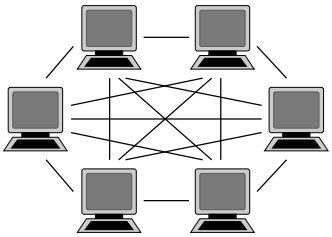


FIG. 2.1 – Topologie point-à-point

aux pannes puisqu'une panne d'un des nœuds scinde le réseau et limite fortement les possibilités de communication.

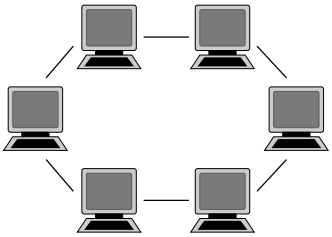


FIG. 2.2 – Topologie en anneau

En pratique, chaque nœud est en fait relié à un répartiteur (le *Multistation Access Unit*, MAU) qui se charge de simuler l'anneau et d'accorder des intervalles de communications à chaque nœud. C'est le principe de l'anneau à jeton (ou *Token Ring*).

iii. Topologie en arbre

Le coût de retransmissions peut être limité en utilisant une topologie en arbre. Le coût de retransmissions se voit donc borné en $2.log(N)$.

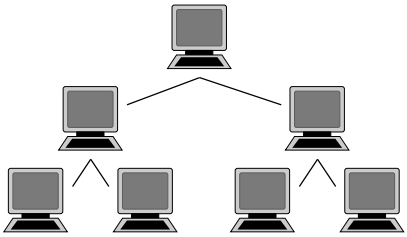


FIG. 2.3 – Topologie en arbre

Le découpage en sous-réseaux lié à cette topologie est particulièrement intéressant.

iv. Topologie en étoile

La topologie en étoile nécessite l'utilisation d'une machine particulière sur le réseau : le *répétiteur* par lequel transitent tous les messages. Dans cette topologie, chaque machine est connectée uniquement au répétiteur dont le rôle consiste à écouter les messages entrants et les répéter sur la liaison du destinataire. Cette stratégie fixe donc le nombre de retransmissions à 2. Cette topologie est beaucoup moins vulnérable aux pannes puisqu'une panne d'un des nœuds n'aura d'impact que sur les communications avec ce nœud. La panne du répétiteur est, quant à elle, beaucoup plus

problématique puisqu'elle rend tout communication impossible. Il est alors possible d'utiliser un second répétiteur qui pourra prendre le relai en cas de panne. Notons également que le répétiteur peut rapidement se transformer en un goulot d'étranglement.

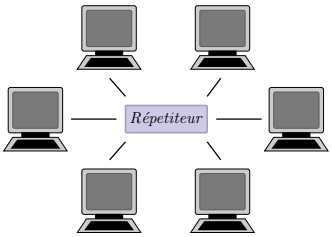


FIG. 2.4 – Topologie en étoile

Dans le cadre de cette thèse, les réseaux utilisés sont les réseau utilisant la topologie en étoile. Ce type de réseau présente notamment l'avantage d'être extrêmement répandu.

v. Topologie en bus

Cette topologie est la plus simple de toutes. Chaque nœud est directement relié à un câble commun à tous les nœuds du réseau. Il existe deux types de bus de communication : les bus *unidirectionnels* qui ne permettent les communications que dans un sens (il est alors nécessaire d'utiliser 2 câbles distincts pour permettre les communications dans les deux sens ou d'utiliser 2 canaux multiplexés) et les bus *bidirectionnels* sur lesquels les données peuvent transiter dans les deux sens mais de façon *non simultanée*.

Cette topologie présente le principal avantage d'être très économique en terme de liens de communications. Elle supporte également assez bien les défaillances puisqu'une panne ne scinde pas le réseau en deux sous-réseaux.
Le principal inconvénient de cette topologie provient de la possible dégradation de performances en cas de charge importante.

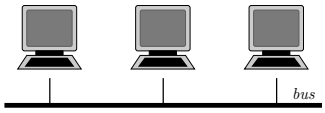


FIG. 2.5 – Topologie en bus

Cette topologie est généralement associée au protocole Ethernet. Plus de détails concernant le protocole Ethernet, ainsi que sur les réseaux de communications en général, peuvent être trouvés dans [141] et [143].

2.1.2 Algorithmes répartis

Les algorithmes pour les systèmes à mémoire répartie reposent souvent sur la notion de site maître (*master*) et de site esclave (*slave*). Un site esclave se contente généralement d'effectuer un certain nombre de calculs en collaboration directe ou non avec un certain nombre de sites esclaves tiers.

Un site maître a pour objectif de gérer le processus dans son ensemble (certaines fois en sus d'une activité équivalente à celle d'un site esclave en terme de charge de calcul). Son rôle est donc

particulier puisqu'il requiert d'avoir une vision globale de l'exécution. Généralement un maître est responsable de l'initialisation et de la terminaison de l'algorithme réparti. Il peut également tenir un rôle de superviseur en effectuant du *monitoring*. En récoltant un certain nombre d'informations sur chaque site esclave, il peut alors prendre un certain nombre de décisions globales concernant par exemple le rééquilibrage de charge. Suivant les algorithmes, un ou plusieurs sites maîtres peuvent cohabiter. Les algorithmes mettant en œuvre plusieurs maître sont généralement basés sur des architectures logiques similaires à la topologie en arbre, avec une notion de hiérarchie entre les maîtres : les feuilles sont les sites esclaves, les autres sites sont les sites maîtres qui vont gérer les sites issus de leur branches.

2.1.3 Communications

Dans les systèmes répartis, les différents processus communiquent entre eux et coopèrent via des échanges de messages.

La bibliothèque de communications MPI s'est imposée comme le standard de librairie de communications pour les applications parallèles à échange de messages dans les systèmes à mémoire répartie.

Nous décrirons ici les fonctionnalités de base proposées par ce standard que nous avons utilisé lors de cette thèse. Nous aborderons tout d'abord les mécanismes des communications point-à-point (a.) puis ceux des communications collectives (b.).

a. Communications point-à-point

Les communications point-à-point mettent en œuvre deux sites distants : un émetteur et un récepteur. Le standard MPI offre différents modes de communication point-à-point et notamment deux grandes catégories de modes de communications : le mode bloquant et le mode non bloquant.

i. Communications bloquantes/non bloquantes

Un processus faisant appel à une primitive de communication bloquante – que ce soit en émission ou en réception – reste bloqué sur l'appel de cette primitive jusqu'à ce que les données reçues ou émises soient sauvegardées. Dans le cas d'un appel à une primitive d'émission, l'exécution du processus reste suspendue jusqu'à ce que les données aient été soit expédiées, soit recopiées dans un *buffer* d'émission. Dans le cas d'un appel à une primitive de réception, l'exécution est suspendue jusqu'à réception complète des données.

Les primitives non bloquantes, quant à elles, rendent la main immédiatement après leur appel. Aucune garantie n'est donc offerte concernant les données, laissant le soin au programmeur d'éviter les éventuelles erreurs d'écrasement de *buffer* d'émission ou de réception. Certaines primitives sont alors offertes pour savoir si un message a été envoyé ou reçu afin de ne pas écraser un *buffer* en cours d'écriture.

ii. Communications synchrones/asynchrones et mode ready

La notion de primitive bloquante ou non bloquante est régulièrement confondue avec la notion de communication synchrone ou asynchrone. Il convient d'éviter cette erreur et de bien distinguer ces deux notions.

Dans les communications en mode synchrone, l'émission d'un message ne se fait pas sans une synchronisation entre l'émetteur et le récepteur. L'émission du message ne débute donc que lorsque le récepteur a acquitté une demande d'émission de l'émetteur. Dans un mode synchrone, lorsqu'un message est émis, un récepteur est donc forcément à l'écoute du message. Une émission en mode synchrone n'est cependant pas nécessairement bloquante.

Cette synchronisation n'existe pas lors d'une communication asynchrone . Dans ce mode, les émissions s'effectuent sans qu'un récepteur ne soit forcément à l'écoute. Une émission en mode asynchrone peut tout à fait être bloquante.

Une communication en mode *ready* nécessite également une synchronisation entre l'émetteur et le récepteur. La différence entre le mode *ready* et la communication en mode synchrone se fait un niveau du déclenchement de la synchronisation. Dans une communication synchrone , c'est l'émetteur qui envoie une requête d'émission au récepteur que ce dernier doit acquitter avant que l'émetteur ne puisse débuter le transfert de son message. Dans une communication en mode *ready*, c'est au récepteur d'initialiser le transfert en envoyant une requête à l'émetteur. Il signale à l'émetteur qu'il est prêt (*ready*) à recevoir un nouveau message.

iii. Communications bufferisées

Lors d'une communication *bufferisée*, le message envoyé n'est pas directement transmit sur le réseau. Il est recopié dans un premier temps dans un *buffer* avant d'être envoyé. La gestion de ces *buffers* (allocation, désallocation, ...) est laissée au soin du programmeur.

iv. Conclusion

Les figures 2.6 et 2.7 présentent les différents modes de communications possible avec MPI. Nous avons illustré ici les mécanimes de communications point-à-point en considérant l'envoi d'un message avec différents types de communication. Pour montrer l'interêt de ces différents types de communication, nous avons présenté l'état d'attente d'un processus par une ligne plus fine (—) que la ligne illustrant l'exécution normale (▬).

Comme on peut le voir sur ces figures, utiliser des primitives non bloquantes peut grandement améliorer l'efficacité d'un algorithme réparti puisque les opérations peuvent – dans certains cas – être effectuées sans attente. Cette situation est présentée dans l'exemple de l'envoi en mode synchrone non bloquant : l'émetteur effectue un appel à la primitive d'envoi puis, comme pour toute communication non bloquante, un appel à la primitive `Wait` est effectué pour vérifier la complétion de l'opération d'envoi et éviter ainsi les écrasement de *buffer*. Ici, l'appel à cette dernière primitive ne sera pas bloquant puisque l'émission du message est déjà effectuée. Ce cas de figure est le cas idéal pour les communications non bloquantes puisqu'il réduit le coût des communications à son plus strict minimum puique le processus peut continuer son exécution pendant la réception du message et ne reste pas bloqué comme c'est le cas en mode bloquant. Ce gain d'efficacité peut être obtenu tant pour le récepteur que pour l'emmetteur lors de l'utilisation de communications non bloquantes.

b. Communications collectives

Dans certains cas, il est utile d'effectuer des communications mettant en œuvre plus de deux nœuds du réseau. Ces communications, appelées communications collectives, se divisent en trois catégories :

1. diffusion d'un message à un ensemble de nœuds,
2. distribution de données à un ensemble de nœuds,
3. récoltes de données d'un ensemble de nœuds.

Nous présentons ici un rapide descriptif de ce type de communication proposé dans le standard MPI.

i. Diffusion

Lors d'une diffusion, un nœud envoie de façon – théoriquement – atomique un *même message* à un ensemble de nœuds. Dans l'exemple présenté sur la figure 2.8, le nœud 0 diffuse un message aux nœuds 1 et 2.

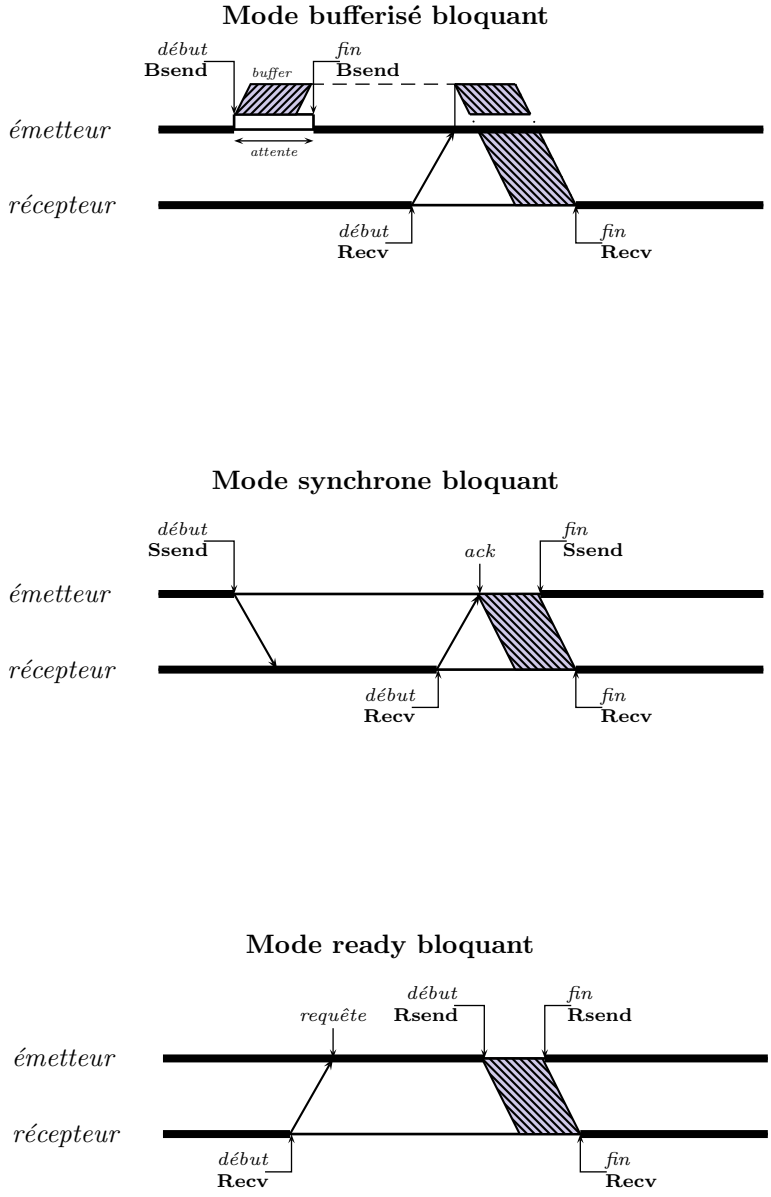


FIG. 2.6 – Communications MPI en mode bloquant

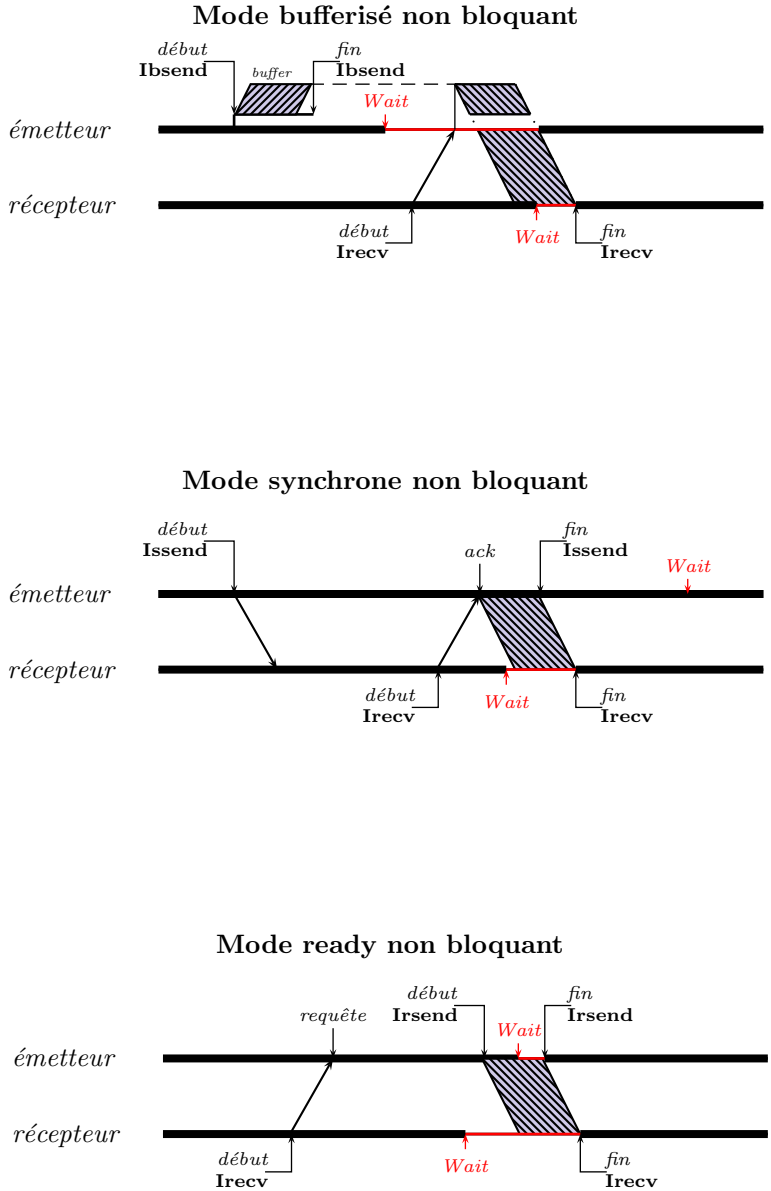


FIG. 2.7 – Communications MPI en mode non bloquant

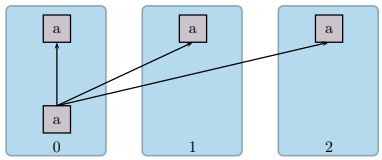
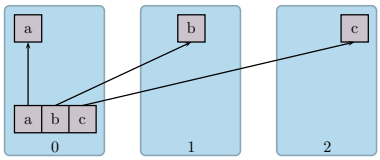


FIG. 2.8 – Opération de diffusion

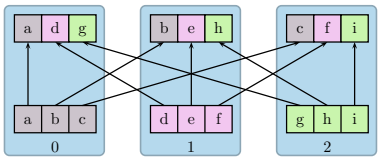
Cette opération est assimilable a une succession d'émission du même message à l'ensemble des nœuds du réseau.

ii. Distribution

Le opérations de distribution consistent à répartir un ensemble de données le plus également possible au sein d'un groupe de nœuds. Deux opérations existent comme l'illustre la figure 2.9. La première est la distribution simple qui distribue un ensemble de donnée d'un nœud dans un ensemble de nœuds.



Distribution simple (*scatter*).



Distribution collective (*all to all*).

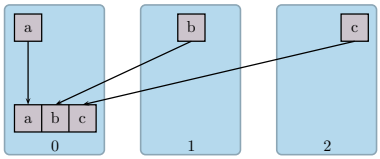
FIG. 2.9 – Opérations de distribution

Dans les opérations de distribution, c'est la primitive qui décide du découpage des données à envoyer.

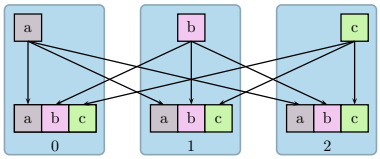
La distribution collective, quant à elle, permet la répartition d'un ensemble de données issu de chaque nœud d'un ensemble vers tous les nœuds de cet ensemble.

iii. Récolte

Les opérations inverses des opérations de distribution sont les opérations de récolte (cf. figure 2.10). Là encore, deux types d'opération sont possibles. La récolte simple consiste à rassembler sur un seul nœud un ensemble de données présentes sur chaque nœud du réseau. Cette opération est l'opération complémentaire de la distribution simple.



Récolte simple (*gather*).



Récolte collective (*all gather*).

FIG. 2.10 – Opérations de récolte

La récolte collective permet quant à elle, de rassembler des données présentes sur chaque nœud du

réseau. La différence de cette opération avec la récolte simple réside dans le fait que ce rassemblement de données est effectué sur chaque nœud.

2.1.4 Synchronisation et détection de la terminaison

a. Synchronisation

Le standard MPI propose une primitive de synchronisation collective appelée, *barrière*, qui permet à un ensemble de nœuds de se synchroniser à un point donné dans l'exécution. Nous présenterons très succinctement cette méthode de synchronisation dans le cadre de systèmes multiprocesseurs pour lesquels le principe de la barrière reste le même.

b. Détection de la terminaison

Un programme réparti est sensé se terminer lorsque plus aucun site ne continu de travailler et/ou que le but final de l'algorithme a été atteint. L'inactivité des sites n'est pas la seule information a prendre en considération puisqu'un site peut être réveillé dès la réception d'un message.

Généralement le site en charge de la détection de la terminaison est le site maître. Détecter la terminaison d'un algorithme réparti requiert donc un minimum de communications pour éviter toute fausse détection de la terminaison qui pourrait entraîner une terminaison prématurée de l'algorithme.

De nombreux algorithmes de détection de la terminaison pour les systèmes à mémoire répartie se basant sur diverses hypothèses ont été proposés dans la littérature. Un certain nombre de ces algorithmes peut être trouvé dans [130]. Nous pouvons également citer l'algorithme de détection de terminaison par vagues de Mattern [128] basé sur des compteurs de messages envoyés et reçus, l'algorithme de visite des sites sur un anneau de Misra [129] ou encore l'algorithme de Safra que nous utilisons dans cette thèse.

L'algorithme de Safra se base sur la différence entre le nombre de messages reçus et envoyés sur un réseau en étoile. Il a été initialement présenté dans [125] et se base sur l'algorithme de Dijkstra [126].
Il consiste à utiliser un compteur c sur chaque nœud. L'envoi d'un message incrémente le compteur tandis que la réception le décremente. Lorsqu'un nœud n souhaite effectué une détection de la terminaison, il envoie un jeton avec la valeur 0 à son voisin qui passera alors par chaque nœud avant de revenir à n. Chaque nœud i conserve le jeton jusqu'à ce qu'il devienne inactif puis le passe à son voisin en l'incrémentant de c. Notons qu'en plus du compteur, chaque nœud ainsi que le jeton ont également un $flag$ qui peut être *noir* ou *blanc*. Lorsqu'un nœud reçoit un message, son $flag$ passe à *noir* (il devient actif). Lorsqu'un nœud transmet le jeton, il passe à *blanc* (il est inactif). Lorsqu'un noeud *noir* reçoit le jeton, il le colore alors en *noir* sinon, le jeton conserve sa couleur. Lorsque n reçoit le jeton à nouveau, il peut conclure à la terminaison si les trois conditions suivantes sont vérifiées :

1. le nœud n est passif et *blanc*
2. le jeton est *blanc* et
3. la somme $jeton + c$ vaut 0.

Dans le cas contraire, n peut lancer une nouvelle détection.

2.1.5 Equilibrage de charge

L'équilibrage de charge est un concept important des systèmes répartis. Cette opération vise à utiliser les ressources disponibles le plus efficacement. Pour cela, il est nécessaire que les différents nœuds du réseau soient actifs de façon la plus continue qui soit. Le but étant de limiter les temps

d'inactivité.

Un mauvais équilibrage peut généralement aboutir à deux comportements plus ou moins dommageables :
– temps d'inactivité importants sur certains nœuds ; ce qui aboutit nécessairement à une perte d'efficacité en terme de temps d'exécution
– surcharge d'un ou de plusieurs nœuds : cette situation est beaucoup plus grave que la précédente puisqu'elle peut empêcher la résolution du problème posé. Imaginons le cas où un problème nécessite 5Go de mémoire pour s'exécuter et que 3 machines de 2Go sont disponibles. Si la charge n'est pas correctement équilibrée et que 3Go sont délégués à un nœud, le processus ne pourra aboutir alors que les ressources disponibles sont théoriquement suffisantes.

En général, l'équilibrage de charge est géré par la (ou les) station maître en fonction des données récoltées sur les différents nœuds.

2.1.6 Cohérence

Le problème de la cohérence est un problème propre aux systèmes répartis. Que l'on parle de cohérence de données, liée à l'ordre dans lequel les données sont accédées, ou de cohérence d'actions liée à l'ordre dans lequel sont exécutées les actions.

Cette cohérence causale est liée à la notion de dépendance entre les actions. Si l'on considère deux actions a et b, on peut alors identifier trois cas possibles :

1. a et b sont indépendantes, auquel cas a et b peuvent être exécutées sans notion d'ordre,
2. a et b sont commutatives, exécuter la séquence $a.b$ ou $b.a$ est alors équivalent
3. a et b sont dépendantes et (par exemple) a doit toujours être exécuté avant b sous peine de rendre la séquence invalide. On parle alors de *précédence causale* entre a et b.

C'est dans ce troisième et dernier cas qu'interviennent les problèmes de cohérences. Pour conserver une cohérence dans les algorithmes, on peut alors essayer de garantir un ordre entre les actions, soit en essayant de respecter la causalité (*via* les horloges vectorielles, par exemple), soit en respectant un ordre total (*via* les horloges de Lamport).

Pour illustrer rapidement les notions d'ordre causal et d'ordre total, considérons un système réparti :
– l'ordre causal implique que si la diffusion d'un message m' dépendait causalement de l'émission d'un message m, alors un processus correct qui délivre m' délivrera m avant m'
– l'ordre total garantit que si un processus correct délivre m avant de délivrer m', alors tous les autres processus corrects délivreront m avant m'.

2.2 Systèmes multiprocesseurs

Dans la section précédente, nous avons abordé la notion de système à mémoire répartie. Le principal objectif de la parallélisation dans ce type d'environnement est d'augmenter les ressources mémoires disponibles pour résoudre un problème donné. On peut également aboutir à des gains de temps mais cet objectif reste secondaire dans ce type de systèmes.

Les systèmes multi-processeurs à mémoire partagée ne permettent pas d'augmenter les capacités mémoires mais proposent d'exécuter des tâches de façon simultanée pour diminuer le temps de traitement.

Nous présenterons ici les principes généraux des systèmes parallèles à mémoire partagée et leurs problématiques.

2.2.1 Parallélisme virtuel et parallélisme réel

Lorsque l'on parle de parallélisme, il convient de distinguer deux types de parallélisme : le *parallélisme virtuel* et le *parallélisme réel* , comme présenté sur la figure 2.11.

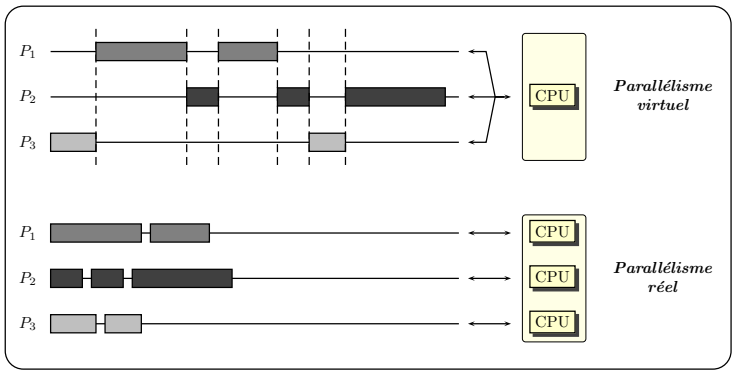


FIG. 2.11 – Parallélisme réel et parallélisme virtuel

Lorsque plusieurs processus s'exécutent sur un *système monoprocesseur multitâche*, on parle de parallélisme virtuel. Dans ce cas de figure, le parallélisme n'est que simulé puisque les processus ne s'exécutent pas *en même temps* mais à *tour de rôle*. Chaque processus se voit donc périodiquement attribuer un *quantum* de temps pour s'exécuter. Cette opération réalisée par un *ordonnanceur* est appelée *ordonnancement*. Cette opération se charge également de sauvegarder et restaurer les contextes d'exécution de chaque processus. Sans rentrer dans la description des différentes politiques d'ordonnancement, citons tout de même les politiques d'ordonnancement FIFO (*First In, First Out*), le *round-robin*, le CFS (*Completely Fair Scheduler*) ou encore l'ordonnancement SJF (*Shortest Job First*). Certaines politiques d'ordonnancement peuvent également gérer des notions de priorités entre les processus.

Les *systèmes multiprocesseurs* , quant à eux, permettent le parallélisme réel. Ces systèmes sont composés du plusieurs processeurs. Des processus peuvent donc s'exécuter parallèlement sur les différents processeurs (cf. figure 2.11). L'utilisation de systèmes multiprocesseurs peut donc aboutir à de formidables gains de temps. Dans ces systèmes, les processus sont placés sur les différents processeurs et peuvent s'exécuter simultanément.

Dans le cas des système monoprocesseur multitâche comme dans les systèmes multiprocesseurs, les différents processus partagent les différents périphériques de la machine. Ils partagent notamment la mémoire vive ainsi que l'espace disque.

2.2.2 Processus lourds et processus légers (*threads*)

Un *processus léger* (*lightweight process*) est un *flot d'exécution* interne à une entité appelée *processus lourd* (*heavyweight process*). Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit *monoprogrammé* : son exécution est réalisée de manière séquentielle par un fil de contrôle (*thread of control*). Par opposition, un processus lourd *multiprogrammé* est un processus lourd contenant plusieurs fils d'exécution : l'exécution est réalisée par ces différents fils de manière potentiellement parallèle.

Chaque processus lourd implique la gestion d'un nouveau contexte d'exécution : nouvel espace d'adressage virtuel, nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous, ...). La création de ce nouvel espace spécifique à un unique

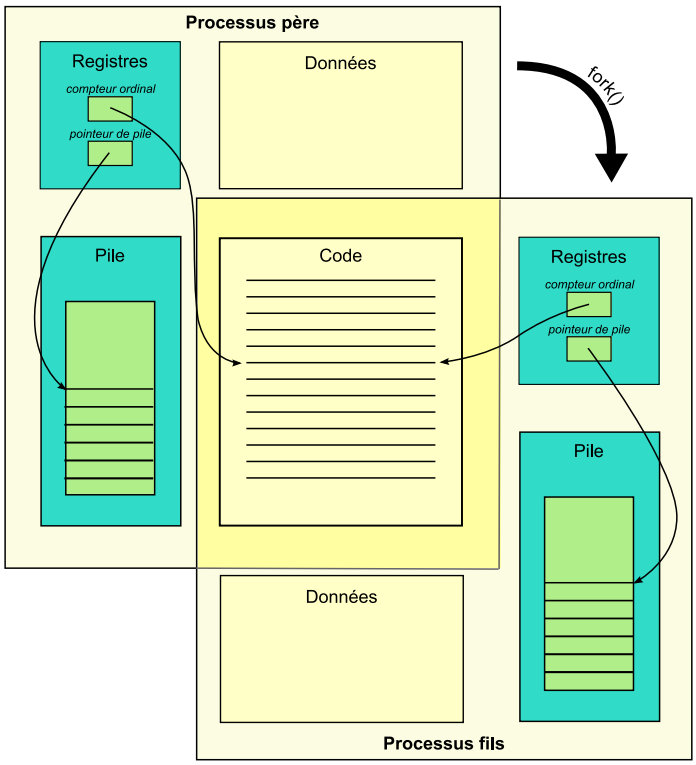


FIG. 2.12 – Duplication d'un processus lourd.

processus lourd offre certaines garanties concernant la protection des données stockées mémoires. Cependant cette création nécessite l'allocation d'un contexte d'exécution ainsi que la duplication de toutes les données du père.
Le qualitatif "léger" est attribué aux processus légers car, comparativement aux processus lourds, ils ne consomment que très peu de ressources : seuls la pile d'exécution et les registres sont alloués lors de la création d'un *thread*. Les autres ressources sont partagées par tous les processus légers s'exécutant au sein du même processus lourd (cf. figure 2.13).

Cette notion de partage de la mémoire est importante. Dans le cas de deux processus lourds, la mémoire est partagée également mais chaque processus ne peut accéder qu'à une zone dans la mémoire qui lui est réservée. Dans le cas d'un processus lourd engendrant plusieurs processus légers, le partage est plus important puisque chaque processus accède à la zone mémoire réservée au processus lourd initial.

2.2.3 Modèles de coopération entre processus

De la même façon que dans les systèmes répartis, les systèmes parallèles se basent sur la notion de coopération entre processus. Dans les systèmes répartis, cette coopération s'effectue à l'aide de messages échangés entre les processus. Ici, la notion de réseau n'est plus présente ; les techniques de communications entre processus ne sont donc pas les mêmes. Elles reposent cependant – pour certaines d'entre elles – sur des concepts très similaires.

Cette synchronisation est extrêmement importante dans le cadre de processus s'exécutant dans une même zone mémoire puisque plusieurs processus peuvent se trouver en *concurrence* d'accès à une variable. Le résultat d'une lecture d'une variable peut être tout à fait incohérent si l'opération

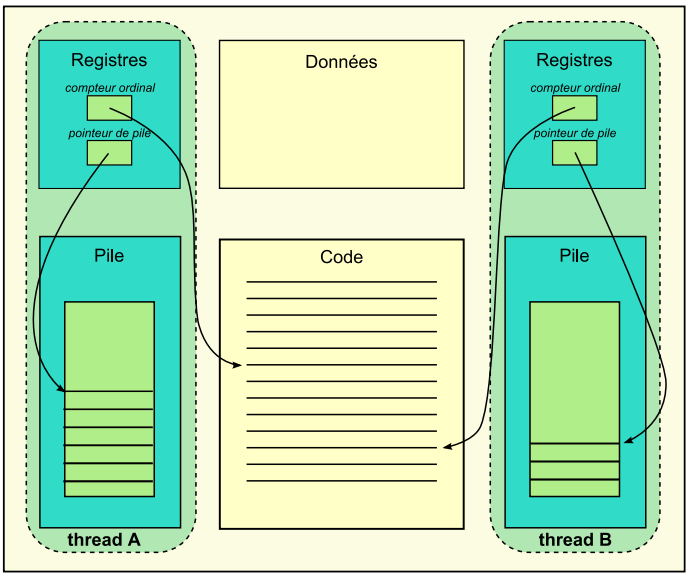


FIG. 2.13 – Processus lourd contenant deux processus légers.

est effectuée en même temps qu'une opération d'écriture sur cette variable par un processus tiers.

a. Barrière

Ce modèle est un mécanisme très simple de synchronisation. Il permet de synchroniser un groupe de processus à un endroit précis du programme. Lorsqu'un processus arrive sur cette instruction, il se trouve bloqué jusqu'à ce que tous les processus du groupe soient arrivés sur cette même barrière. Une fois les processus arrivés sur cette instruction, ils sont alors tous débloqués et peuvent reprendre leur exécution. Une barrière de synchronisation peut donc permettre de garantir que tous les processus d'un groupe ont tous passé un point spécifique du programme.

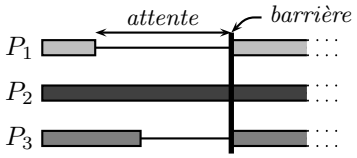


FIG. 2.14 – Barrière de synchronisation

Ce mécanisme de synchronisation est également utilisé en programmation répartie et défini dans le standard MPI.

b. Exclusion mutuelle

L'exclusion mutuelle est la forme de synchronisation entre processus la plus simple et probablement l'une des plus utilisées dans les systèmes à mémoire partagée. L'exclusion mutuelle sert à protéger des zones de code critiques – appelées sections critiques – qui ne peuvent être exécutées que par un seul processus à la fois (accès en écriture à des données partagées, ...).
L'interblocage (p. 61) est l'un des risques majeurs de l'exclusion mutuelle.

c. Rendez-vous

Le rendez-vous est une méthode de synchronisation qui permet à plusieurs processus d'effectuer une action simultanément. Lorsque tous les processus concernés par un rendez-vous sont arrivés à l'instruction du rendez-vous, le rendez-vous peut s'effectuer.

d. Producteur/consommateur

Le modèle de producteur/consommateur (fig. 2.15) est un modèle de communication extrêmement utilisé dans les systèmes à mémoire partagée. On distingue ici deux types de comportement : le *producteur* qui transmet un certain nombre de données et le *consommateur* qui les récupère. Les données sont écrites dans un tampon – qui peut être de n'importe quel type : file, liste, ... – par le producteur. Cette opération peut être bloquante en écriture si la taille du tampon est bornée et que le tampon est plein. De la même façon, le consommateur reste bloqué sur un tampon vide. Une donnée ne peut être lue qu'une seule fois, une fois lue, elle est retirée du tampon. Comme toute structure partagée, le tampon doit être protégé par des verrous pour éviter des accès concurrents. Cependant, un algorithme sans verrou pour un tampon représenté sous forme de liste simplement chaînée a été proposé dans [123].

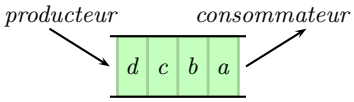


FIG. 2.15 – Producteur/consommateur

e. Lecteur/rédacteur

Le modèle de lecteur/rédacteur – connu également sous le nom de *read-write lock pattern* – est un modèle de synchronisation particulièrement intéressant pouvant considérablement améliorer l'efficacité d'un programme multithreadé. Pour présenter ce modèle, prenons un exemple très simple. Considérons n threads notés $t_1, \ldots, t_n$. Les threads $t_2, \ldots, t_n$ effectuent une succession de calculs nécessitants la valeur la plus récente d'une variable v mise à jour régulièrement par t_1. Une première solution serait de protéger cette variable en y autorisant l'accès uniquement en exclusion mutuelle. On remarque cependant que l'accès en lecture concurrent de plusieurs processus ne présente aucun risque puisque la donnée n'est pas modifiée par les opérations de lecture. Il n'y a donc aucun risque pour un processus de lire une valeur incohérente. L'accès en exclusion mutuelle tant en lecture qu'en écriture est donc extrêmement lourde et limite fortement le parallélisme.

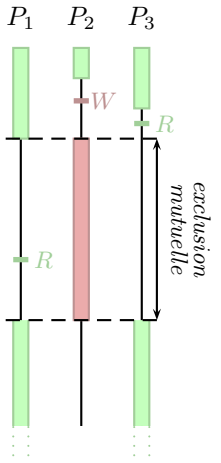


FIG. 2.16 – Lecteur/rédacteur

Le modèle lecteur/rédacteur permet justement à plusieurs processus d'accéder en lecture à une donnée de façon simultanée tout en garantissant l'accès exclusif à cette donnée en écriture. Ainsi lorsqu'un processus veut accéder en écriture sur une donnée, il est bloqué tant qu'un autre processus y accède, que ce soit en lecture ou en écriture. Lorsqu'un processus veut accéder à la donnée en lecture, il reste bloqué si un processus y accède déjà en écriture mais peut y accéder s'il n'y a que des accès en lecture. Sur la figure ci-contre, 3 processus P_1, P_2 et P_3 accèdent initialement à une même donnée en lecture. Une fois que P_2 a relâché en lecture, il cherche à l'acquérir en écriture(W). Il est alors mis en attente jusqu'à ce que tous les lecteurs aient relâché le verrou. L'accès en écriture se fait alors de façon exclusive plaçant en attente les processus P_1 et P_3 qui redemandent l'accès en lecture(R). Ils seront tous les deux réveillés dès que P_2 aura terminé son accès en écriture.

Pour éviter tout risque de famine (p. 62) d'un rédacteur, il convient également de bloquer tous les nouveaux lecteurs lorsqu'un rédacteur est en attente d'accès à la donnée.

2.2.4 Mise en œuvre de la synchronisation

Nous présentons ici les mécanismes permettant la mise en œuvre de la synchronisation interprocessus.

a. Attente active

Dans une stratégie basée sur l'attente active, un processus bloqué est chargé de vérifier, de façon répétée, la valeur sur laquelle il est bloqué. Cette vérification est effectuée à l'aide d'une boucle infinie dans laquelle le processus effectue uniquement son test de sortie de boucle.

Un processus effectuant une attente active continue à consommer les ressources cpu c'est pourquoi cette technique de synchronisation n'est utilisée que dans certains cas très spécifiques. Il est généralement préférable d'utiliser des mécanismes d'attente passive.

b. Attente passive

Les mécanismes de synchronisation basé sur l'attente passive ont pour but de résoudre les problèmes d'efficacité de l'attente active : ici, un processus en attente ne monopolise pas les ressources cpu.

i. Les sémaphores

Un sémaphore est un mécanisme de synchronisation fonctionnant comme une barrière. Un sémaphore s comporte un compteur entier n_s ainsi qu'une file d'attente de processus f_s. Les sémaphores proposent deux opérations notées P et V (cf. figure 2.17). Ces opérations doivent être réalisées de

```
P (s)                                  V (s)
1  n_s ← n_s − 1;                      1  if n_s < 0 then
2  if n_s < 0 then                     2     p ← pop(f_s);
3     set_process(p, bloqué);          3     set_process(p, actif);
4     f_s ← f_s ∪ p;                   4  end if
5  end if                              5  n_s ← n_s + 1;
```

FIG. 2.17 – Algorithmes des opérations P et V sur un sémaphore s.

manière atomique et nécessitent donc d'être protégées par une technique d'exclusion mutuelle.

L'opération P consiste en une tentative de franchissement du sémaphore, cette opération peut être bloquante si le franchissement est suspendu. L'opération V, quant à elle, permet le relâchement du sémaphore par un processus.

Le principe d'un sémaphore est assez simple. Il consiste a autoriser son franchissement par, au plus, n_s processus simultanément. La valeur de n_s est affectée lors de la création du sémaphore. Lorsqu'un processus franchi le sémaphore, il décrémente le compteur n_s (fig. 2.17, opération P, ligne 1). Lorsque le compteur devient négatif, les processus restent en attente de l'autorisation de franchissement (fig. 2.17, opération P, ligne 3) et sont placés dans la file d'attente f_s.

Lorsqu'un processus relâche le sémaphore (opération V), si un ou plusieurs processus sont en attente du verrou, un processus peut alors être libéré de cette file et franchir le sémaphore. Dans la majeure partie des cas, la sélection du processus à débloquer se fait simplement dans l'ordre FIFO. Ceci permet d'éviter les risques de famine (p. 62). Il est cependant possible d'utiliser d'autres algorithmes d'élection – basés par exemple sur un mécanisme de priorités.
La figure 2.18 présente un algorithme sans famine mettant en œuvre le modèle lecteur/rédacteur. Soit s un verrou de type lecteur/rédacteur, on associe une variable $readers_s$ initialisée à 0 qui représente le nombre de lecteurs actuellement en accès, un sémaphore $read_s$ d'exclusion mutuelle (ie. initialisé à 1) utilisé par les lecteurs visant à protéger la variable $readers_s$, un sémaphore $write_s$ d'exclusion mutuelle utilisée par les rédacteurs et un sémaphore sem_s d'exclusion mutuelle permettant de garantir l'accès exclusif en écriture et l'absence de famine.

```
Read_Lock (s)
1  P(sem_s);
2  P(read_s);
3  readers_s ← readers_s + 1;
4  if readers_s = 1 then P(write_s);
5  V(read_s);
6  V(sem_s);

Read_UnLock (s)
1  P(read_s);
2  readers_s ← readers_s − 1;
3  if readers_s = 0 then V(write_s);
4  V(read_s);

Write_Lock (s)
1  P(sem_s);
2  P(write_s);

Write_UnLock (s)
1  V(write_s);
2  V(sem_s);
```

FIG. 2.18 – Sémaphores pour l'implémentation du modèle lecteur/rédacteur sans famine

ii. Les verrous

Les verrous (*mutex* POSIX) sont une simple spécialisation des sémaphores. Un verrou est un sémaphore d'exclusion mutuelle. Sa valeur est donc initialisée à 1.

iii. Les événements

La synchronisation par événement est un autre outil de synchronisation entre processus comparable au mécanisme d'interruption. Le principe est le suivant : un processus – ou groupe de processus – peut se mettre en attente d'un signal. Ce signal sera alors envoyé par un processus tiers.

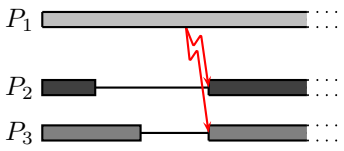


FIG. 2.19 – Synchronisation par signaux (*broadcast*)

Lorsque plusieurs processus sont en attente d'un signal, deux opérations de déblocage sont possibles : une débloquant un processus parmi le groupe de processus en attente et une autre permettant le déblocage de tous les processus (*broadcast*). Le choix du processus débloqué lorsque l'on utilise la première opération n'est pas déterministe. De plus, un signal envoyé mais non récupéré est *définitivement perdu*. Cette dernière caractéristique est particulièrement importante, elle induit notamment qu'il est impossible de savoir si un signal envoyé sera récupéré par un processus.

iv. Les messages

De façon similaire aux communications dans les réseaux de machines, il est possible de synchroniser des processus au sein d'une même machine via des échanges de messages.

Les *pipes* (ou tuyaux) Unix sont un exemple de cette méthode de synchronisation. On distingue alors deux types de *pipes* :
- les pipes non nommés, permettant la communication entre deux processus qui partagent le même environnement,
- les pipes nommés, qui permettent à deux processus totalement indépendants (c'est-à-dire sans lien de parenté) de communiquer.

Les *pipes* fonctionnent dans un seul sens et le message est automatiquement retiré du *pipe* une fois qu'il a été lu.

v. Les moniteurs

Le mécanisme des moniteurs cherche à proposer une méthode de synchronisation plus structurée que celle proposée par les sémaphores. Ceci facilite notamment la compréhension et l'écriture des schémas de synchronisation.
Un moniteur est assimilable à un objet comportant plusieurs variables d'états internes *privées* ainsi que plusieurs procédures accessibles depuis l'extérieur et modifiant les valeurs des variables d'états.

```
1   moniteur tampon {
2     queue q;
3     signal empty,full;
4
5     procedure synchronized deposer (in m : message){
6       if full(q) then
7         wait(full);
8       end if
9       enqueue(q, m);
10      notify(empty);
11    }
12
13    procedure synchronized retirer (out m : message){
14      if empty(q) then
15        wait(empty);
16      end if
17      dequeue(q, m);
18      notify(full);
19    }
20  }
```

```
producteur ()
1   ...
2   msg ← produire();
3   tampon.deposer(msg);
4   ...
```

```
consommateur ()
1   ...
2   tampon.retirer(msg);
3   lire(msg);
4   ...
```

FIG. 2.20 – Utilisation d'un moniteur pour le modèle du producteur/consommateur

Ces procédures contiennent des opérations qui vont bloquer ou réveiller les processus utilisant le moniteur. Chaque procédure est exécutée en exclusion mutuelle pour garantir la cohérence des variables d'états du moniteur.

c. Conclusion

Il existe une grand variété de mécanismes de synchronisation, chacun répondant à des problèmes différents. Les sémaphores et les verrous sont des mécanismes de bas niveau alors que les mécanismes tels que les moniteurs sont des constructions de plus haut-niveau que l'on retrouve notamment dans le langage Ada.

2.2.5 Problèmes courants

L'utilisation des différents mécanismes de synchronisation inter-processus introduits précédemment permet d'assurer la validité d'un programme multithreadé en réglementant le parallélisme. L'absence de ces mécanismes dans un programme multithreadé entraîne généralement un certain nombre d'incohérences – voire d'erreurs – lors de l'exécution. Il est en effet possible qu'un processus essaie de récupérer la valeur d'une variable alors qu'au même moment, un autre processus la modifie.

Les mécanismes de synchronisation sont donc tout à fait indispensables dans ce type de programmes. Toutefois, la mise en œuvre de ces mécanismes peut s'avérer compliquée et donc source de nombreuses erreurs et notamment des problèmes de vivacité (ou *liveness*).

a. Interblocages

L'interblocage (ou *deadlock*) est le problème de vivacité le plus courant : un processus p_1 se met en attente d'une action d'un processus p_2 alors que p_2 est lui-même en attente d'une action de p_1. Dans le cas de la figure 2.21, considérons deux verrous m_1 et m_2 et supposons que p_1 et p_2 aient chacun exécutés leur première instruction. Si p_1 exécute sa seconde instruction pendant que p_2 exécute la sienne, p_1 et p_2 sont alors en interblocage : l'action attendue par p_1 est la libération du verrou m_2 détenu par p_2. Dans le même temps, p_2 est en attente de la libération de m_1 détenu par p_1. Les deux processus s'attendent donc mutuellement sans possibilité de déblocage.

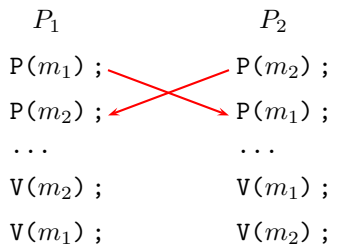


FIG. 2.21 – Exemple d'algorithme pouvant entraîner un interblocage

Plusieurs méthodes existent pour gérer les problèmes d'interblocage. Ces méthodes sont au nombre de 4 : prévention, évitement, détection et validation. Les 4 paragraphe suivants présentent succinctement ces 4 méthodes.

i. Prévention des interblocages

L'objectif de la prévention d'interblocages est de garantir qu'un interblocage ne pourra jamais se produire.

Dans la prévention d'interblocages, un processus doit déclarer toutes les ressources dont il aura besoin lors de l'exécution d'une séquence d'instructions. La séquence ne pourra s'effectuer que si et seulement si toutes les ressources requises sont disponibles et si le système peut garantir qu'aucune de ces ressources ne sera requise par une séquence en cours.

Cette stratégie garantit ainsi l'absence d'interblocages et permet donc d'éviter le recours aux mécanismes de *rollback* ou de redémarrage dûs aux interblocages. Elle souffre cependant de deux désavantages majeurs : tout d'abord, la pré-réservation de ressources peut fortement réduire la

concurrence – notamment pour des séquences utilisant un grand nombre de ressources –, de plus, évaluer la sûreté d'une séquence rajoute un surcoût en temps potentiellement élevé.

ii. Evitement des interblocages

Dans les stratégies d'évitement d'interblocage, il n'est pas nécessaire d'effectuer une pré-réservation de ressources. Lorsqu'un processus veut accéder à une ressource non disponible, il se met en attente. Deux possibilités de réveils sont alors possible : soit la donnée devient disponible et l'exécution se poursuit normalement, soit l'attente de la ressource s'arrête au bout d'un certain temps (grâce à un système de *timeout*). Dans les second cas, le processus considère alors qu'il est potentiellement en interblocage et libère donc un certain nombre de ressources qu'il détient et recommence son exécution à partir d'un certain point *via* un mécanisme de *rollback*.

Cette solution permet donc d'éviter les interblocages mais les *timeout* peuvent dégrader les performances soit en permettant des attentes trop longues, soit en détectant de faux interblocages si les temps d'attentes sont trop courts.

iii. Détection des interblocages

Les stratégies de détection d'interblocage permettent aux processus de se mettre en attente de ressources sans limitation de durée. Des interblocages peuvent alors se produire : ils doivent donc être détectés puis résolus.

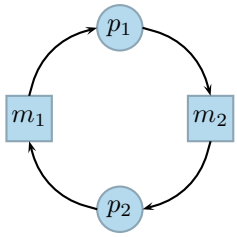


Fig. 2.22 – Graphe de dépendances.

La détection de ces interblocage se fait au moyen de graphes dirigés de dépendances. La figure 2.22 présente le graphe de dépendance obtenu lors d'un ordonnancement aboutissant à un interblocage pour l'algorithme présenté figure 2.21. Les processus sont modélisés par des cercles et les ressources par des carrés. Lorsqu'un processus acquiert une ressource, un arc partant de la ressource vers ce processus est ajouté au graphe. Lorsqu'un processus demande une ressource, un arc partant de ce processus vers la ressource est ajouté. Un interblocage est finalement détecté lorsqu'un cycle apparaît dans le graphe.
Lorsqu'un cycle est détecté, plusieurs politiques de résolution sont possibles : soit en tuant un processus, soit en réattribuant une ressource, ...

Plus de détails et d'exemples d'algorithmes concernant ces trois premières stratégies peuvent être trouvés notamment dans [127] et [131].

iv. Validation

La dernière stratégie permettant de détecter et de supprimer les interblocages est le sujet même de cette thèse : la validation de programme – ou de spécification. Le principe est simple : valider formellement le programme – ou sa spécification – en vérifiant qu'aucun interblocage n'est possible quelle que soit l'exécution considérée.

Aucun mécanisme additionnel de gestion d'interblocages n'est donc nécessaire dans le programme ; il n'y a donc aucun surcoût en mémoire ou en temps (si ce n'est le temps nécessaire à la validation du programme).

Cette technique n'est cependant applicable qu'aux systèmes finis et souffre du problème de l'explosion combinatoire présenté précédemment.

b. Famine

La famine (*starvation*) est un autre problème de la programmation concurrente. La famine se traduit par l'incapacité d'un processus à accéder à une ressource ; même si celui-ci est prêt à

y accéder. Ce problème peut apparaître lorsque des mécanismes de priorités sont utilisées : un processus à faible priorité peut alors voir ses requêtes d'accès à une ressource infiniment repoussées au profit de processus prioritaires.

c. Terminaison prématurée

Dans certain cas, suite à un certain ordonnancement, il est possible qu'un processus se termine de façon prématurée empêchant ainsi le programme d'aboutir.

d. Endormissement

Lorsqu'un processus se met en attente d'une action d'un processus (comme un signal par exemple) il est possible qu'il ne soit jamais réveillé – notamment dans le cas d'un signal envoyé avant que le processus ne se soit mis en attente du signal. On parle alors d'endormissement, ce qui aboutit au mieux à une non terminaison du programme.

2.3 Conclusion

Nous avons vu dans ce chapitre quelques caractéristiques des systèmes multiprocesseurs et des systèmes répartis. Ces deux types de systèmes se basent sur la notion de coopération entre plusieurs processus. Ces processus s'exécutant soit sur des machines distinctes soit sur une même machine en partageant le même espace mémoire.

Dans le cas de la parallélisation via les systèmes répartis, le but principal est d'augmenter la mémoire disponible de manière à permettre la résolution de problèmes qui ne pourraient être envisagée sur une machine simple. Un objectif secondaire peut également être la réduction du temps d'exécution dans le cas de programmes permettant un degré de parallélisation suffisants et dans lesquels la granularité des travaux effectués en local est suffisamment importante pour compenser les temps de communications (transmission, traitement, ...) inhérents à ce type de systèmes.

Pour les systèmes à mémoire partagée, l'objectif est unique et porte sur le temps d'exécution. Ici, on cherche à effectuer un certain nombre de tâche de façon simultanée afin de réduire le temps total d'exécution.

Le développement d'applications, dans l'un ou l'autre de ces types de systèmes, présentent des problématiques particulières tels que l'interblocage, la détection de la terminaison, ... Ces problématiques rendent le développement complexe puisque plusieurs exécution successives n'aboutissent généralement pas au même enchaînement d'actions ; il est alors complexe de bien identifier des cas d'exécution d'erreur au vue de la difficulté de ré-éxecuter plusieurs fois le même enchaînement d'actions.

Nous avons présenté plusieurs mécanismes et concepts liés à ces types de systèmes que nous avons utilisé lors de cette thèse.

Deuxième partie

Model checking parallèle et réparti

Chapitre

3

Model checking réparti

Sommaire

Ce chapitre a pour objectif de présenter les principes de base du model checking parallèle dans un environnement à mémoire répartie. Nous y présenterons tout d'abord le principe général (3.1) de l'approche, puis nous porterons une attention plus particulière aux différents mécanismes à modifier pour prendre en compte ce nouvel environnement.

Nous commencerons par présenter la problématique liée au partitionnement de l'espace d'état (3.2). Nous détaillerons les différentes approches que nous avons considérées pour aborder le partitionnement de réseaux de Petri colorés avant de les évaluer et de les comparer sur un certain nombre de modèles.

Nous nous pencherons ensuite sur l'adaptation des méthodes de représentation et de compression des états (3.3). Nous détaillerons tout particulièrement les stratégies adoptées pour résoudre le problème de l'adaptation de la technique des Δ-*markings* dans un environnement réparti.

Nous aborderons enfin le problème de la construction du rapport d'erreur (3.4) ainsi que de la détection de la terminaison de l'algorithme d'exploration du graphe d'accessibilité (3.6) avant de discuter des techniques d'équilibrage de charges.

A chaque fois, nous effectuerons une rapide présentation de l'état de l'art puis nous décrirons les solutions que nous avons apportées aux différents problèmes.

3.1 Principe général

Le model checking réparti repose sur la notion de répartition de l'espace d'état. Le but est de tirer parti de la quantité de mémoire ainsi que de la puissance de calcul offerte par le regroupement d'un ensemble de machines de travail (*Network Of Workstations – NOW*). Le principe est donc simple, chaque nœud – ou site – du réseau est responsable du stockage d'une partie de l'espace d'état.

Les techniques de model checking que nous utilisons stockent tous les états dans la mémoire tout au long de l'exploration contrairement à certaines techniques qui peuvent être amenées à supprimer des états de la mémoire soit parce que l'on sait qu'ils ne seront plus utiles, soit parce que l'on se place dans le cadre de stratégies de vérification probabilistes. Dans ces cas-là, le gain en puissance de calcul est un élément aussi important que la quantité de mémoire. Dans notre cas de figure, c'est la quantité de mémoire qui pose essentiellement problème ; le gain en puissance de calcul est intéressant mais reste un atout secondaire.

Dans l'approche séquentielle classique, le parcours de l'espace d'état peut se faire de deux façons différentes : (1) parcours en largeur (*Breadth First Search* – BFS) ou (2) recherche en profondeur (*Depth First Search* – DFS). La première solution est la moins fréquemment utilisée car elle limite l'efficacité de certaines méthodes et notamment des techniques d'ordre partiel. Généralement, l'exploration se fait donc en profondeur. La méthode de recherche est une méthode récursive présentée ci-contre (fig. 3.1), utilisant la notion de pile de recherche en profondeur. Chaque nouvel état visité est ajouté sur la pile puis est retiré lorsque tous ses successeurs ont été visités. Cette pile est particulièrement importante, c'est notamment elle qui permet d'appliquer efficacemment les réductions d'ordre partiel. Nous verrons qu'elle est également la source d'un certain nombre de problèmes lors du passage à un environnement à mémoire répartie.

```
DFS (s)
1   if s ∉ Visited then
2      Visited ← Visited ∪ {s}
3      stack ← stack ∪ {s}
4      for s' ∈ Successor(s) do
5         DFS(s')
6      end for
7      stack ← stack \ {s}
8   end if
```

FIG. 3.1 – Recherche en profondeur (DFS)

L'ensemble *Visited* correspond ici à l'espace des états accessibles déjà rencontrés. La méthode *Successor* calcule pour un état donné, l'ensemble de ses états successeurs.

Lors du passage à un environnement à mémoire répartie, la procédure de recherche en profondeur doit subir quelques modifications. Ces modifications sont présentées sur la figure 3.2. L'idée est assez simple : pour chaque état s' visité par un nœud n_i, il faut vérifier si s' doit être stocké sur n_i ou sur un autre nœud. Initialement, chaque nœud est responsable d'une partition de l'espace d'état. Le partitionnement de l'espace d'état doit être connu de tous les nœuds. Pour cela, chaque nœud a recours à une unique *fonction de partition* dont nous aborderons les caractéristiques plus en détail un peu plus loin dans ce chapitre. Cette fonction de partition associe à un état s, la partition à laquelle s "appartient".

```
DFS (s)
1   if s ∉ Visited then
2     Visited ← Visited ∪ {s}
3     stack ← stack ∪ {s}
4     for s' ∈ Successor(s) do
5       let owner ← partition(s')
6       if owner = id then
7         DFS(s')
8       else
9         send(s',owner)
10      end if
11    end for
12    stack ← stack \ {s}
13  end if
```

FIG. 3.2 – Procédure DFS en environnement réparti

Lorsque un nœud n_i visite un état s', il doit vérifier s'il est responsable du stockage de la partition à laquelle appartient s'. Ce test est effectué à la ligne 6 de l'algorithme présenté à la figure 3.2. Le calcul de la partition à laquelle appartient l'état s' est effectué à la ligne 5. Si l'état doit être stocké sur n_i, la procédure continue de la même façon qu'en séquentiel. Dans le cas contraire, l'état est envoyé à son propriétaire. Le nœud n_i considère alors s' comme totalement exploré. L'état s' est donc dépilé et l'exploration peut se poursuivre.

Comme nous l'avons précisé précedemment (2.1.2), les algorithmes répartis distinguent généralement deux types de nœuds : le nœud maître et les nœuds esclaves. C'est également l'approche utilisée par la majorité des model checkers répartis. C'est l'approche que nous avons utilisée lors de nos travaux.

Le nœud maître (que nous appelerons également *manager*) est responsable de l'initialisation de la recherche ainsi que de la détection de la terminaison. Comme présenté à la figure 3.3, l'initialisation de la recherche s'effectue tout simplement en calculant l'état initial et en l'envoyant au nœud responsable de son stockage (lignes 1 à 3).

```
manager ()
1   let s0 ← initial()
2   let owner ← partition(s0)
3   send(s0, owner)
4   while not termination do
5     // Réception de statistiques
6     Queue ← Queue ∪ receive()
7     termination ← detect_termination(Queue)
8   end while
9   broadcast(termination)
```

FIG. 3.3 – Procédure principale du nœud maître

Dans Cyclades (la version parallèle du model checker Helena) le *manager* n'est responsable d'aucune partition : il ne participe donc pas, à proprement parlé, à l'exploration. Son rôle consiste uniquement à initialiser et gérer la terminaison de l'algorithme. Ce choix est un choix de simplicité et de facilité d'évolution. Dans un processus de vérification réparti simple, le *manager* se contente de piloter le processus global. Les évolutions que nous avons commencé à introduire dans Cyclades et celles que nous souhaitons ajouter, obligent le *manager* à gérer éventuellement plusieurs processus de vérification simultanément. Pour une meilleure réactivité, il nous a semblé que limiter la charge du *manager* était une bonne solution. Toutefois, dans le cas simple, il est tout a fait possible

de placer le *manager* sur un nœud également utilisé par un nœud esclave. Cette stratégie permet donc de ne pas "gâcher" un nœud.

Le comportement du *manager* est alors simple : une fois l'état initial envoyé, il se met en attente de messages entrants qui lui permettront de détecter la terminaison. Une fois celle-ci détectée, un message de terminaison est diffusé sur le réseau.

Le comportement des nœuds esclaves est présenté à la figure 3.4. Tant qu'un message de terminaison n'a pas été reçu, les nœuds esclaves sont en attente de messages entrants.

```
worker ()
1   while not termination do
2     Queue ← Queue ∪ receive()
3     if termination received then
4       termination ← true
5     else
6       while Queue ≠ ∅ do
7         let s ← pop(Queue)
8         DFS(s)
9       end while
10      if no incoming messages then
11        send(statistics, manager)
12      end if
13    end if
14  end while
```

FIG. 3.4 – Procédure principale des nœuds esclaves

Ces messages entrants sont généralement de deux types : message de terminaison envoyé par le *manager* – auquel cas le processus peut terminer – ou message d'exploration envoyé par un autre nœud esclave. Ces messages d'exploration sont en fait des états qu'il faut stocker puis explorer à l'aide de la DFS.

La procédure principale commune au maître et aux esclaves est présentées à la figure 3.5.

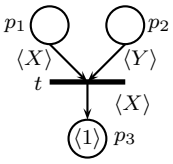
```
main ()
1   Queue ← ∅
2   Visited ← ∅
3   termination ← false
4   id ← getId()
5   if id = manager then
6     manager()
7   else
8     worker()
9   end if
```

FIG. 3.5 – Procédure principale commune à tous les nœuds

La méthode *getId* permet de récupérer l'identifiant du nœud et donc de reconnaître le *manager* des esclaves. Cette méthode est définie dans le standard MPI (`Get_Rank`).

Remarque Nous nous limiterons ici au model checking réparti explicite. Plusieurs travaux sur le model checking réparti ont été menés dans d'autres contextes comme le model checking symbolique [98, 96, 97], le model checking basé sur des probabilités [74], le model checking réparti pour le

μ-calcul [72], le model checking réparti temporisé [69, 73] ou enfin une approche pour le model checking basé sur le franchissement arrière d'actions [70] ce qui peut, pour le type de modèles que nous analysons, aboutir à une infinité de marquages. Considérons par exemple une simple transition t prenant en entrée 2 jetons X et Y issus respectivement de places p_1 et p_2 et produisant un unique jeton X dans une place p_3. Le franchissement arrière de cette transition peut alors générer autant de marquages que la cardinalité du domaine de couleur de Y. Si ce domaine de couleur est l'ensemble des entiers naturels $\mathbb{N}$, le nombre de marquages possibles est donc infini.



3.2 Partitionnement de l'espace d'états

L'efficacité du processus de vérification en environnement réparti est fortement dépendant du partitionnement de l'espace d'état. Autant les objectifs du partitionnement sont aisés à appréhender – degré de localité le plus élevé possible et bonne répartition de la charge – autant la recherche d'une bonne méthode de partitionnement est ardue.

Le principal soucis rencontré concerne la difficile prévision du comportement de l'espace d'état : nombre d'états, nombre d'arcs, degré de concurrence, efficacité des réductions (d'ordre partiel notamment), ...

Nous verrons que plusieurs approches ont déjà été proposées pour aborder ce problème (3.2.3) ; certaines visant une distribution équitable de l'espace d'état sans se soucier du nombre de communications générées où du degré de localité de la répartition, d'autres, au contraire, essaient de s'appuyer sur le modèle pour tenter d'apporter des réponses plus adaptées et plus spécifiques aux différents modèles.

Nous présenterons ici un bilan de ces différentes méthodes. Nous nous attacherons également à montrer comment ces méthodes peuvent être modifiées pour être adaptées aux réseaux de Petri (3.2.4) colorés ou non.

Comme nous l'avons précisé précédemment, le principe du model checking réparti consiste à associer un sous-ensemble de l'espace des états accessible à chacun des nœuds du réseaux. Soit S l'ensemble des états accessibles et S_i un sous-ensemble d'états accessibles assigné au nœud i, on a alors :

$$S = \bigcup_{i=0}^{n-1} S_i$$

où n est le nombre de nœuds disponibles pour la vérification. Notons également que le partitionnement vérifie la propriété suivante :

$$\bigcap_{i=0}^{n-1} S_i = \emptyset$$

Cette seconde propriété stipule qu'aucun état ne peut appartenir à plusieurs partitions. Elle nous permet alors de garantir qu'aucun état ne sera dupliqué sur plusieurs nœuds du réseau.

3.2.1 Fonction de partition

Lorsqu'un nœud explore un état il utilise alors une *fonction de partition* sur cet état pour déterminer à quelle partition – et donc à quel nœud– il appartient. Cette fonction de partition doit être connue de chaque nœud dur réseau. Pour cela, elle est définie à la compilation et est la même pour chaque nœud.

Trouver une fonction de partition d'un algorithme de vérification distribué est un problème complexe. Dans le cas d'un problème de répartition ou de parallélisation classique, une bonne fonction

de partition est une fonction qui répartit les données à traiter de façon uniforme. Dans le cas d'algorithmes pouvant générer un grand nombre de communications entre les noeuds – comme c'est le cas pour le model checking– il est également intéressant d'essayer de répartir les données de façon à ce que la répartition limite autant que possible les communications et préserve/garantisse ainsi un bon degré de localité.

3.2.2 Notion de *cross-transition*

On utilise la notion de *cross-transition* pour désigner une transition qui, une fois franchie, génère un état appartenant à un autre noeud (donc à une autre partition) (cf. figure 3.6). Ces transitions particulières résultent directement du partitionnement et génèrent des échanges de messages. Un des objectifs du partitionnement va alors être de limiter le nombre de ces transitions.

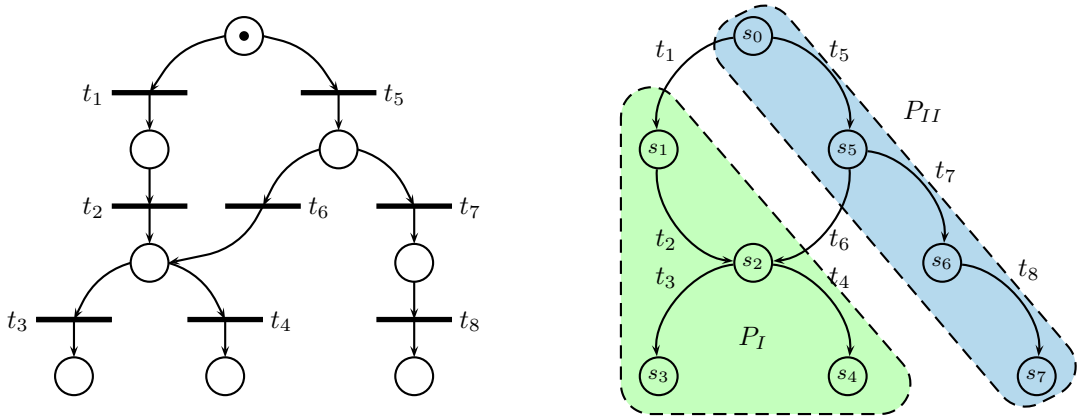


FIG. 3.6 – *Cross-transitions*

Sur cette figure, les transitions t_1 et t_6 sont des *cross-transitions* puisqu'elles génèrent des arcs partant de la partition P_{II} vers la partition P_I. Une *cross-transition* est une transition qui va alors générer des communications lors de l'exploration de l'espace d'état.

3.2.3 Etat de l'art

Nous présentons ici les principaux algorithmes de partitionnement de l'espace d'état existants. Ces algorithmes peuvent se séparer en deux grandes catégories : les algorithmes de partitionnement dits uniformes qui visent à répartir les états de la façon la plus régulière entre chaque partition et les algorithmes qualifiés de structurels qui s'appuient sur une analyse statique du modèle pour essayer de trouver une fonction de partition qui limitera le nombre de communications en limitant le nombre de *cross-transitions*.

a. Partitionnement uniforme

La première implémentation d'un model checker réparti a été présentée dans [83]. Le *verifier* Murφ est un outil de vérification de protocoles de communications, sa version parallèle a pour objectif de paralléliser le processus de vérification dans un environnement à mémoire répartie.

Concernant la méthode de partitionnement, les auteurs ont utilisé une approche très simple en se basant sur une simple fonction de hachage. Il réutilisent simplement la fonction de hachage interne (h_{sp}) utilisée pour l'espace d'état :

$$f(s) = h_{sp}(s) \mod n$$

où n représente le nombre de nœuds. L'objectif de la fonction h_{sp} étant d'obtenir une répartition équilibrée des états dans la table de hachage représentant l'espace d'état, la fonction de partition f aura donc vraisemblablement le même objectif. Les expérimentations effectuées confirment cette idée et présentent une répartition extrêmement efficace en terme d'équilibre de charge – d'états – même pour de larges espaces d'états et/ou pour un nombre élevé de nœuds utilisés.

La principale limite de cette approche réside dans le faible degré de localité obtenu suite au partitionnement : la probabilité qu'un état s_j, successeur de s_i stocké sur le nœud i, appartienne également à i est proche de $1/n$. Pour de petites valeurs de n (2, 3, 4, ...), le résultat est déjà faible ; il devient un facteur extrêmement limitant dès que n dépasse ces faibles valeurs.

Ce faible degré de localité dégrade bien évidemment les performances du processus puisque le nombre de communications s'en voit fortement augmenté. Il peut également fortement limiter l'efficacité de certaines méthodes comme les réductions d'ordre partiel ou les Δ-*markings* que nous présenterons dans la suite de cette thèse.

b. Partitionnement structurel

Pour essayer de répondre au problème du partitionnement basé sur une simple fonction de hachage qui génère des partitions offrant une très faible degré de localité, plusieurs approches ont été proposées. Ces approches, toutes très similaires, se basent sur la structure d'un état et par là même, sur la structure du modèle ; c'est pourquoi nous parlons ici de partitionnement structurel.

La méthode a été initialement présentée dans [76] puis reprise pour être adaptée à Promela dans [80]. Le principe est le suivant : pour un système donné, un état global s peut être vu comme un vecteur d'états locaux (états des processus, des variables, ...). Si l'on applique la fonction de partition sur une partie de ce vecteur plutôt que sur le vecteur en entier, on peut alors augmenter de façon significative le degré de localité du partitionnement. En effet, un changement de partition ne sera effectué que lorsque la partie utilisée pour la répartition sera modifiée. L'exécution d'actions modifiant le reste du vecteur ne modifiera pas la partition.

Considérons par exemple un état s_1 défini comme suit :

$$s_1 : \{p_1, < (0, 1, [1, 3, 4] >\} + \{p_2, < 0, 2, [3, 5] >\}$$

et un de ses successeurs s_2 défini par :

$$s_2 : \{p_1, < (0, 1, [1, 3, 4] >\} + \{p_3, < 0, 3, [3, 5] >\}$$

L'état s_1 correspond alors au marquage des places p_1 et p_2. Le jeton dans p_1 est constitué des 3 valeurs dont un type de haut-niveau (le tableau $[1, 3, 4]$). Supposons que le franchissement d'une transition t exécutable à partir de s_1 aboutisse à l'état s_2. Le franchissement de cette transition a modifié le contenu de la place p_2 (en en retirant le jeton) et de la place p_3 (en y ajoutant un nouveau jeton). Si l'on considère le marquage de la place p_1 comme sous ensemble pertinent pour le partitionnement, le franchissement de t ne modifie alors pas la partition puisque le marquage de p_1 reste inchangé ; s_1 et s_2 appartiennent alors à la même partition. Seul le franchissement d'une transition modifiant le contenu de la place p_1 pourra éventuellement aboutir à un changement de partition.

Cette méthode s'avère particulièrement efficace en terme de degré de localité. Son efficacité dépend cependant du sous-ensemble du vecteur d'états locaux considéré pour le partitionnement. C'est justement là que le problème se pose puisque tous les formalismes ne sont pas égaux devant le choix du sous-ensemble. La solution proposée dans [80] est relativement simple, elle consiste à sélectionner un sous-ensemble pertinent relatif à l'évolution d'un type de processus. Cette sélection est aisée lorsque le modèle est spécifié à l'aide du langage Promela [119] où la notion de processus est explicite. Cette sélection est beaucoup plus ardue pour d'autres formalismes où la notion de processus n'est pas explicite (ce qui est notamment le cas pour les réseaux de Petri).

La solution proposée dans [76] repose donc sur une intervention de l'utilisateur pour effectuer le partitionnement. Cette approche est basée sur l'utilisation d'un arbre AVL[142] appelé *control set*(cf.figure 3.7). Cet arbre est créé lors d'une phase d'initialisation de l'algorithme puis dupliqué sur chaque nœud. A partir de cet arbre, chaque nœud peut retrouver la partition à laquelle appartient un état. Chaque feuille de cet arbre représente une partition.

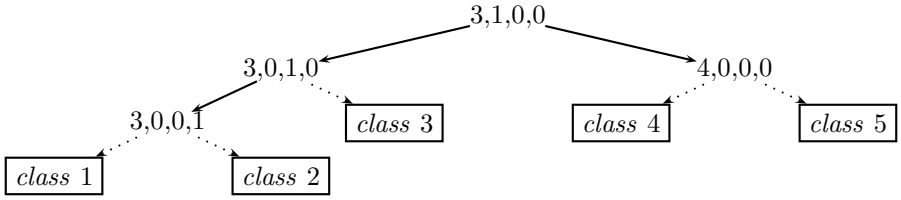


FIG. 3.7 – Exemple de *control set*

Une approche plus automatisée de cette solution et ne nécessitant pas d'intervention de l'utilisateur a été présentée dans [77]. Elle se base une initialisation du *control set* par plusieurs séries d'explorations aléatoires dans l'espace d'état. Ces séries d'exploration peuvent alors donner une certaine idée du comportement du modèle.

Pour les réseaux de Petri, l'adaptation de la méthode consiste à ne considérer que le marquage d'un sous-ensemble de places du réseau pour effectuer le partitionnement. Cette approche a été présentée dans [82] sans toutefois apporter de réponse à la question fondamentale de la méthode de sélection de ce sous-ensemble de places.

3.2.4 Partitionnement pour les réseaux de Petri

Comme nous l'avons vu précédemment, le principe du partitionnement structurel pour les réseaux de Petri consiste à sélectionner un sous-ensemble de places pour effectuer le partitionnement. Le principe étant assez clair, reste le principal problème : comment choisir ce sous-ensemble de places.

Nous présenterons ici trois approches (a., b. et c.) permettant d'effectuer la sélection de ce sous-ensemble.

a. Cas 1 : utilisation d'informations structurelles

Dans le cas où il est possible d'obtenir un ensemble d'informations structurelles sur le modèle – et notamment sur l'identification de comportement de processus – soit par une analyse, soit par des informations directement contenues dans le modèle, on peut alors facilement isoler un certain nombre de sous-ensembles potentiellement pertinents pour effectuer le partitionnement.

Si l'on se replace dans le contexte de la plate-forme Quasar, l'objectif est de permettre la vérification de propriétés sur des programmes concurrents écrits en Ada. Une des étapes du processus de Quasar[1] consiste à traduire un programme Ada en un réseau de Petri. Un certain nombre d'informations concernant le réseau généré peuvent donc être précisées. Il est notamment possible de spécifier qu'une place représente une variable globale, un état d'un processus ou encore une variable locale.

A l'aide de ces informations, il est alors possible d'identifier plus facilement les places correspondant à un type de processus. On peut alors sélectionner un sous-ensemble de places correspondant à un processus et ainsi effectuer un partitionnement se rapprochant de celui présenté dans [80].

[1]Nous reviendrons plus en détails sur le fonctionnement de Quasar dans le chapitre 7.

b. Cas 2 : détection des points de synchronisation

Lorsque l'on se place hors du contexte de la plate-forme Quasar, le problème de la sélection du sous-ensemble de place devient beaucoup plus complexe.

La première approche que nous avons considérée consiste à identifier les points de synchronisation dans le réseau. L'intuition derrière ce choix est que les points de synchronisation sont généralement des actions qui sont potentiellement bloquantes. Lorsqu'un processus arrive sur une transition de synchronisation, il peut alors se retrouver bloqué ; les seules actions franchissables seront alors des transitions qui ne modifieront pas la partition. Ce choix permettra alors de partitionner l'espace en permettant un meilleur degré de localité, tout en isolant une partie de l'entrelacement ce qui peut être intéressant pour les méthodes d'ordre partiel que nous aborderons plus loin dans cette thèse.

Un point de synchronisation est généralement modélisé par une transition avec plusieurs arcs entrants. Le choix des places pertinentes pour le partitionnement se fera donc sur les places en sortie. Et plus exactement sur un sous-ensemble des places en sortie des transitions de synchronisation.

Pour détecter les points de synchronisation, il n'est pas suffisant de considérer uniquement le nombre d'arcs entrants d'une transition. L'accès à une variable locale, par exemple, ne doit pas être considéré comme une action de synchronisation puisqu'elle ne limite jamais le franchissement de la transition. On considère alors différents *patterns* que nous illustrons à la figure 3.8.

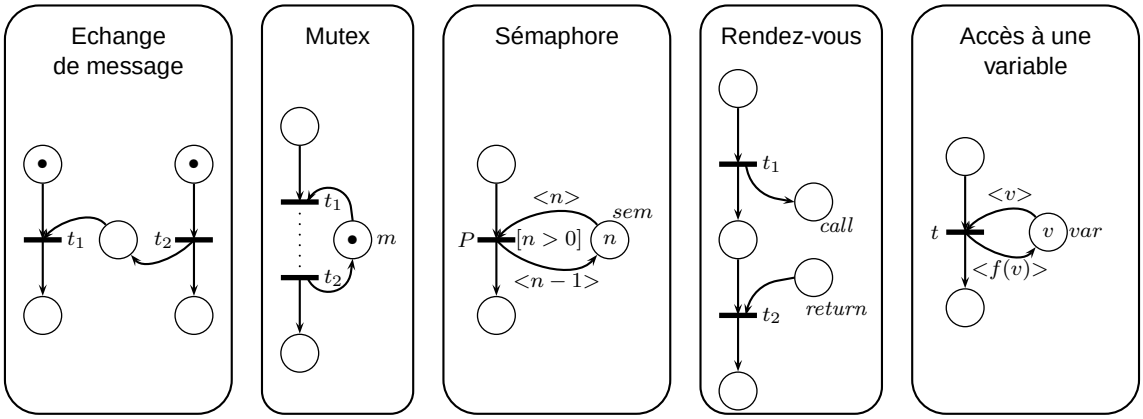


FIG. 3.8 – Quelques exemples de réseaux de Petri.

Dans le cas de l'échange de message, la lecture est modélisée par une transition avec deux arcs entrants : un pour la tâche, un pour la lecture du message (transition t_1). Dans le cas du mutex comme dans celui du rendez-vous, une seule transition est considérée comme transition de synchronisation. Pour le mutex c'est la transition pour laquelle le processus récupère le verrou, pour le rendez-vous c'est la transition permettant à l'appelant de recevoir la réponse (en plus de celle permettant au serveur de recevoir la requête). Le seul cas particulier dans lequel une transition entrante ne doit pas être considérée comme une transition de synchronisation concerne l'accès à une variable locale. Dans ce dernier cas, le pattern correspondant est constitué d'une transition qui récupère un jeton dans une place et le remet dans cette même place sans aucune garde. Ce cas de figure n'est donc pas considérer lors de la recherche de points de synchronisation.

c. Cas 3 : détection automatique de processus

i. Détection de cycle

La détection des points de synchronisation pose toutefois un certain nombre de problèmes dans certains cas dont notamment le cas de modèles tels que ceux de la distribution – qui sont des modèles académiques – ou dans le cas de modèles sur lesquels des réductions structurelles avant la génération ou sur le réseau de Petri ont été appliquées et se sont révélées être particulièrement efficaces.

Dans ces cas là, la majorité des transitions sont des transitions de synchronisation puisque la plupart des actions locales ont été atomisées au sein d'une même transition. Dans ces cas là, la détection de points de synchronisation risque fort d'aboutir à une sélection de toutes les places du réseau. On peut alors décider de n'en sélectionner qu'une partie arbitrairement. Une seconde possibilité consiste à essayer de détecter des comportements cycliques.

En effet, les processus ayant généralement des comportements cycliques, on peut alors essayer de retrouver certains types de processus en détectant les cycles de transitions dans le réseau de Petri à analyser.

Pour ce faire, on essaie alors de détecter les cycles de transitions dans le réseau de Petri. Ces cycles peuvent alors correspondre à différents types de processus mais dans certains cas, ils peuvent être totalement incohérents. Ils permettent cependant d'aboutir à un meilleur degré de localité. Une fois un cycle sélectionné, on considère alors comme sous-ensemble de places pertinent les places présentes sur le cycle.

ii. Calcul de flots colorés

Récemment, nous avons implémenté le calcul des semi-flots colorés [19] dans Helena (nous reviendrons sur ces travaux dans le chapitre 8). Les semi-flots permettent d'obtenir des informations précieuses sur le réseau de façon statique. Il nous permettent entre autre de déterminer des comportements de processus tels que ceux déterminer en effectuant une détection de cycle. Cependant, les cycles détectés ici sont beaucoup plus pertinents qu'un certain nombre de cycle détectés avec la précédente méthode.

Nous n'avons cependant pas encore ajouté cette fonctionnalité à Cyclades.

3.2.5 Réduction des communications

Dans tous les cas de partitionnement, il est intéressant de réduire le nombre de communications générées lors de l'exploration de l'espace d'état. Nous présentons ici trois techniques permettant de réduire ces communications, la première étant une technique très répandue dans les systèmes répartis.

a. Message buffering

Cette méthode consiste simplement à agréger un certain nombre de message en un seul. Dans notre cas, on permet à plusieurs états d'être envoyés dans un seul message. Cette méthode ne réduit donc pas le nombre de données échangées mais permet de réduire les communications.

Il est cependant nécessaire d'adapter légèrement l'algorithme en conséquence en vidant régulièrement les *buffers* d'émission pour éviter tout risque d'attente trop importante des nœuds. Généralement, les *buffers* sont vidés de façon périodique même s'ils n'ont pas atteint la taille maximale. Ils sont également tous vidés lorsqu'un nœud passe au repos, ceci afin d'éviter tout risque de non terminaison de l'algorithme.

b. Children look-ahead

La méthode du *children look-ahead*, introduite dans [81], a pour objectif de limiter le nombre de communications générées par une fonction de partition. Cette méthode est illustrée par la figure 3.9. La répartition obtenue est présentée sur la figure de gauche : les états s_0 et s_2 appartiennent à n_i tandis que s_1 appartient à n_j. Dans cette configuration, l'algorithme ordinaire fonctionne comme présenté sur la figure du milieu : n_i explore s_0 puis ses successeurs – ici s_1 –, comme s_1 n'appartient pas à la partition gérée par n_i, ce dernier envoie l'état au nœud concerné. Le successeur de s_1 n'appartenant pas à n_j, un deuxième message est donc nécessaire pour envoyer s_2 sur n_i. Pour essayer de limiter les communications, la méthode du *children look-ahead* consiste à explorer les successeurs de tous les états même ceux appartenant à une autre partition. De cette façon, dans la configuration proposée, s_2 sera bien exploré par n_i et n'aura pas besoin d'être ré-envoyé par n_j (cf. figure de droite).

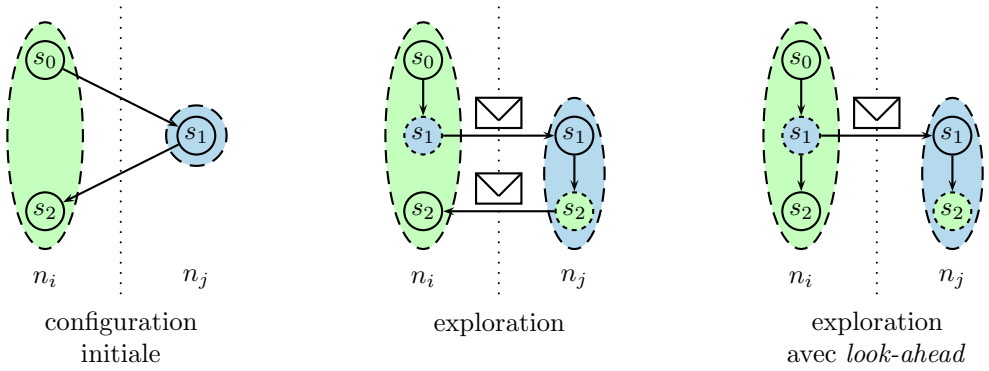
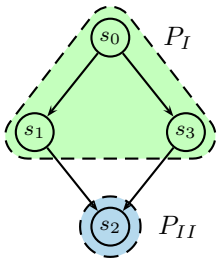


FIG. 3.9 – Exemple de *children look-ahead*

Cette méthode permet donc de limiter le nombre de communications mais engendre un surcoût parfois non négligeable en terme d'exploration puisque pour chaque état envoyé, une double exploration est effectuée. Le coût de cette double exploration peut cependant être réduite dans le cas de partitionnement structurel puisque l'on peut limiter le *look-ahead* au franchissement des *cross-transitions*. En effet, si l'on reprend l'exemple présenté à la figure 3.9, lorsque n_i visite s_1, il effectue une exploration des successeurs de s_1 à la recherche d'éventuels états dont il aurait la charge. Ces états là sont alors forcément issus de transitions modifiant éventuellement le partitionnement, à savoir les *cross-transitions*. Toute autre transition ne modifiant pas la partition, l'exploration des successeurs de s_1 par n_i peut alors se réduire aux seules *cross-transitions* franchissables à partir de s_1.

c. State-caching

La technique du *state-caching* permet de réduire le nombre de communications en réduisant le nombre de données échangées. Plus précisément, cette méthode permet de réduire les envois de requêtes redondantes. Considérons la configuration présentée sur la figure ci-contre. L'exploration de s_1 puis de s_3 va aboutir à un envoi redondant de l'état s_2. Pour éviter ce type de communications inutiles, chaque état envoyé est conservé dans un cache d'émission. Un état n'est donc envoyé que s'il n'est pas déjà présent dans le cache. Cette méthode peut offrir de formidables réductions du nombre de communications quelle que soit l'efficacité du partitionnement effectué. Nous avons évalué l'apport du *state-caching* au regard du nombre de communications engendrées et du temps d'exécution sur deux modèles. Les évaluations sont présentées sur la figure 3.10, elles ont été obtenues en utilisant un partitionnement basé sur la détection de cycles offrant un bon degré



de localité. Les résultats obtenus pour les deux modèles sont assez similaires. Les courbes ont effet les mêmes aspects. Pour les deux modèles, on peut observer un très fort taux d'arcs générants des communications. Lorsque le nombre de nœuds utilisés tend vers 10, on avoisine généralement les 90%. Si l'on se penche sur la courbe du taux de *hit* dans le cache, on observe des valeurs assez élevée, surtout pour des valeurs assez faibles de n. Ce résultat est assez logique puisque plus le nombre de nœuds augmente, plus la possibilité pour un nœud de visiter le même état est réduite puisque la taille de la partition qui lui est attribuée diminue lorsque le nombre de nœuds augmente. On remarque ensuite que le nombre d'arcs générant des communications est fortement réduit lors de l'utilisation du *state-caching*. Là encore, lorsque le taux de *hit* dans le cache diminue, le nombre de communications augmente. On remarque enfin que l'utilisation du cache a permet de réduire de façon très observable temps d'exécution. Ceci permet alors d'appréhender beaucoup plus facilement le coût des communications.

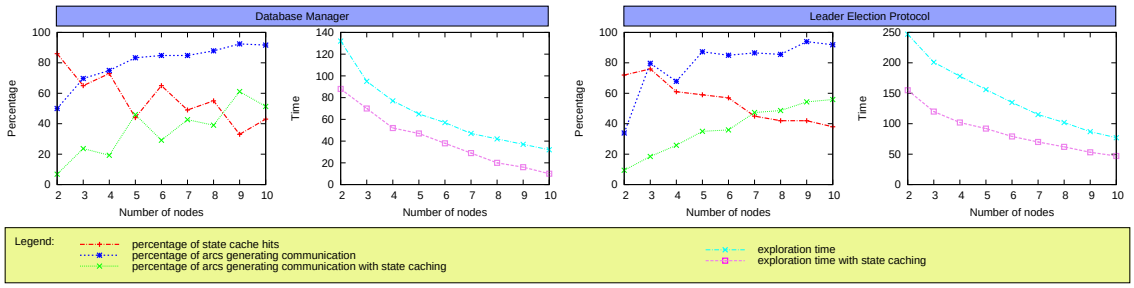


FIG. 3.10 – Evaluation du *state-caching*

3.2.6 Evaluations

Nous avons évalué l'efficacité des différentes stratégies de partitionnement sur différents modèles à l'aide de Cyclades. A chaque fois, nous avons évalué l'efficacité de la répartition à l'aide de différents critères. Ces résultats sont présentés sur la figure 3.11.

L'efficacité d'une fonction de partitionnement peut être analysée au regard du nombre de communications générées ainsi que de l'équité du partitionnement. Le nombre de communications dépend directement du degré de localité obtenu par le partitionnement.

Le premier critère d'évaluation concerne donc le degré de localité obtenu par les différentes stratégies de partitionnement. Ce critère étant le plus important, il est reporté en grisé sur les graphes de la figure 3.11. Cette valeur correspond en fait à la probabilité qu'un successeur s' d'un état s appartienne à la même partition que s. Cette valeur est à mettre en relation directe avec le second critère présenté : le nombre de communications générées lors de l'exploration. L'objectif du partitionnement structurel étant notamment de réduire le nombre de communications il semble important de pouvoir étudier ce critère. Ici, ce nombre en pourcentage d'arcs explorés ayant généré des communications.

Le troisième critère concerne l'équilibrage de la répartition obtenu pour chacune des stratégies. C'est l'un des objectif des fonctions de partition : partitionner l'espace d'état en partitions de taille équivalentes. Ce critère est alors évalué par le biais de l'écart-type obtenu en terme de nombre d'états stockés pour chaque nœud par rapport à une répartition optimale (ie. où chaque nœud est en charge du stockage de $\mathcal{S}/n$ états où $\mathcal{S}$ représente le nombre total d'états et n le nombre de nœud utilisés pour le stockage). Ceci nous permettra notamment de vérifier que l'augmentation potentielle du degré de localité n'a pas été obtenue au détriment de l'équité de la répartition.

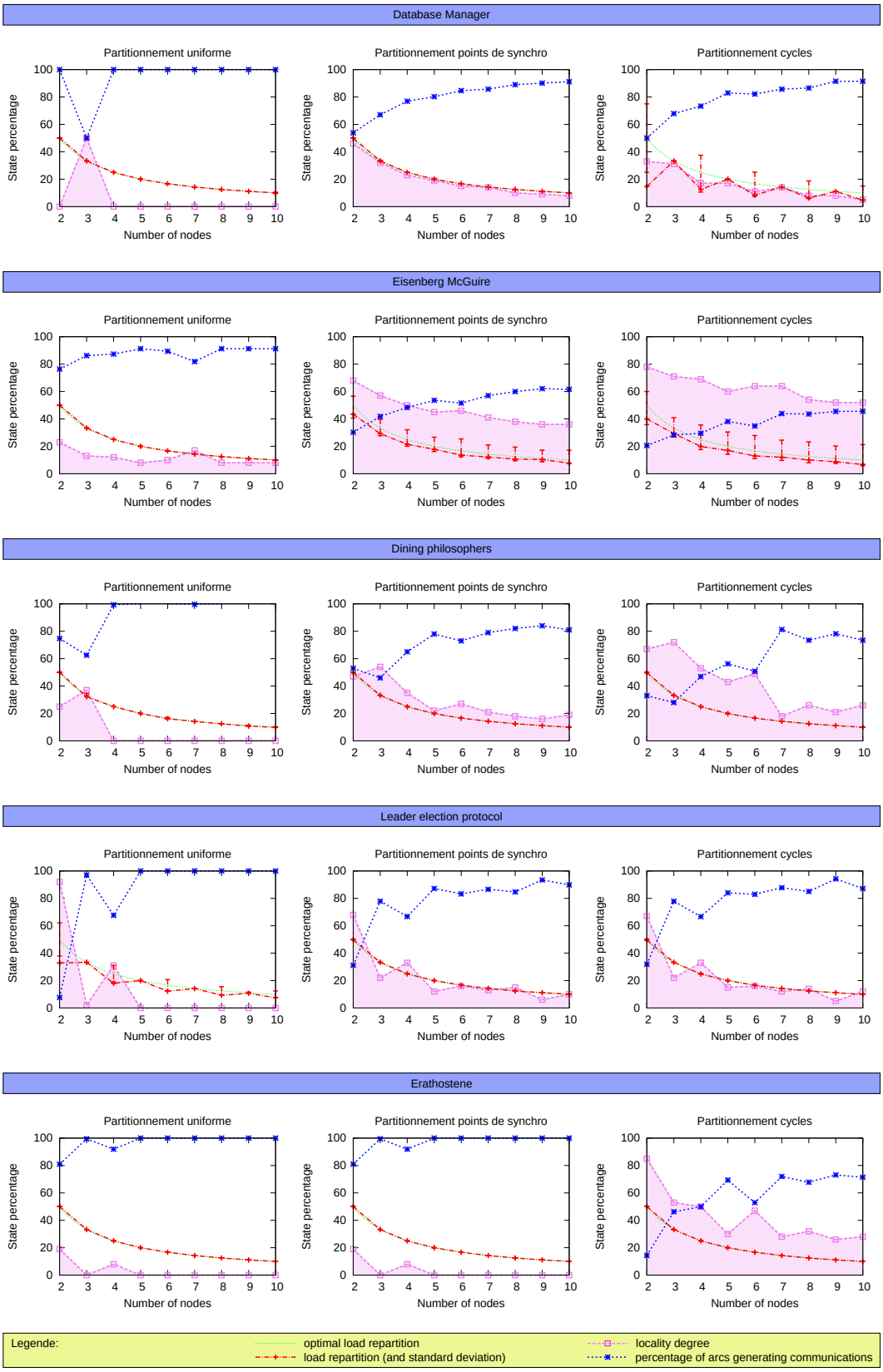


FIG. 3.11 – Evaluation de l'écart-type et du degré de localité pour le partitionnement

Pour cette valeur, nous avons également reporté les tailles maximales et minimales des partitions obtenues.

Notons enfin que seule la technique de partitionnement utilisant l'outil Quasar n'a pas été évaluée. La raison est assez simple ; cette stratégie a pour but de spécifier directement sur le modèle les places correspondant à un type de processus. Ces processus ont généralement un comportement cyclique ; ils seront donc également détectés lors de l'approche basé sur la détection de cycles. La principale différence se situe ici dans le choix parmi les différents cycles ou processus détectés. Dans le cas du partitionnement assisté par Quasar, seuls les processus sont spécifiés, dans le cas d'une détection de cycles, un certain nombre de cycles supplémentaires et parfois non pertinents seront découverts. Dans ces deux stratégies, le choix d'un cycle parmi l'ensemble des cycles détectés se fait de façon assez aléatoire. Généralement, il est nécessaire que ce cycle fasse apparaître un nombre de *cross-transitions* qui ne soit ni trop grand, ni trop faible. Plus le nombre de *cross-transitions* est élevé et plus la répartition risque d'être similaire à celle obtenue est utilisant le partitionnement uniforme. Si l'on choisit un cycle engendrant un faible nombre de *cross-transitions*, on peut alors aboutir à de gros déséquilibres de charge. Dans tous les cas, avec ou sans Quasar, le choix d'un cycle parmi l'ensemble des cycles reste un problème même si les chances d'effectuer un choix pertinent sont augmentées en utilisant Quasar.

Considérons alors les résultats obtenus pour le modèle d'Eisenberg & McGuire ; dans le cas du partitionnement uniforme, on observe que la répartition de la charge (ie. du nombre d'états) est extrêmement proche du cas optimal (les deux courbes sont d'ailleurs confondues). Ceci est le cas pour la plupart des modèles étudiés. Le pourcentage d'arcs générant des communications est tout de suite très élevé même s'il reste en moyenne inférieur à celui observé pour les autres modèles. Dans le cas général, on aboutit très vite à un taux de 100%, c'est-à-dire que chaque état exploré aboutit à un changement de partition ! Le nombre de communications a alors tendance à exploser. Enfin, le taux de localité observé pour Eisenberg & McGuire reste assez faible (environ 15%) ce qui reste même assez élevé en comparaison des autres modèles.

Dans le cas du partitionnement basé sur la détection de points de synchronisation, on observe tout de suite un bien meilleur degré de localité. Ceci est toutefois obtenu en contrepartie d'un équilibrage de charge moins efficace puisque cette fois-ci, la courbe est en dessous de la courbe optimale. On observe également quelques écarts de charges. Notons enfin, que parallèlement à l'augmentation du degré de localité, on observe naturellement une baisse du taux d'arcs générant des communications.

Enfin, dans le cas d'un partitionnement basé sur une détection de cycle, l'efficacité en terme de localité s'accentue au prix d'une répartition légèrement moins efficace. Le pourcentage d'arcs générant des communications passe alors toujours sous la barre des 50% contre les 90% du partitionnement uniforme grâce à la limitation des *cross-transitions*.

Notons que le réseau de Petri modélisant l'algorithme d'Eisenberg & McGuire est un réseau de 22 places et 25 transitions qui, une fois réduit, comporte 17 places pour 20 transitions. Les autres modèles analysés, qui sont également des modèles académiques, sont beaucoup plus petits et comportent en général moins d'une dizaine de places et de transitions. Ceci rend alors les possibilités de sélection des *cross-transitions* beaucoup plus limitées. C'est pourquoi on peut notamment observé deux comportements assez spécifiques : sur le modèle du protocole d'élection, le partitionnement obtenu est sélectionnant les transitions identifiées comme point de synchronisation est rigoureusement identique à celui obtenu suite à une détection de cycles, et sur le modèles du crible d'Eratosthene, on observe que le partitionnement uniforme et le partitionnement basé sur la détection de points de synchronisation aboutissent également aux mêmes résultats. Ceci est en partie lié aux tailles des réseaux et aux réductions structurelles utilisées. Dans le cas du protocole d'élection, les places séléctionnées sont les mêmes pour les deux stratégies de partitionnement structurel. De la même façon, les réductions structurelles utilisées sur le modèle d'Eratosthene ont fonctionné de telle façon que toutes les transitions sont considérées commme des points de synchronisation. Toutes les places

sont alors sélectionnées pour le calcul des partition et l'on retombe alors sur un partitionnement identique à celui obtenu pour le partitionnement uniforme.

Dans le cas général, une rapide observation nous permet de voir que le degré de localité est extrêmement faible dès que l'on utilise le partitionnement uniforme. Très rapidement ce taux passe sous les 10% et fréquemment autour de 0%. Dans la majeure partie des cas, l'utilisation d'une stratégie de partitionnement basé sur une analyse structurelle permet d'aboutir à un meilleur degré de localité même si l'équilibrage peut être assez dégradé (cf. modèle Eisenberg & McGuire).

Le meilleur degré de localité obtenu dans 3 cas sur 5 en utilisant la détection de cycles s'explique principalement par le fait que moins de places sont considérées pour le partitionnement. On aboutit alors à des degré de localité parfois extrêmement élevé, notamment pour le modèle Eisenberg & McGuire même s'il est obtenu au détriment de l'équilibrage.

Dans la pratique les modèles comportants un nombre suffisemment élevé des places et transitions permet un plus grand choix de sélection de places comme sous-ensemble pertinent. On peut alors véritablement aboutir à un partitionnement plus efficace. Il faut cependant noter que le risque de déséquilibre de charge est alors plus grand.

3.3 Représentation de l'espace d'état

Dans cette partie, nous aborderons les éventuels impacts de la répartition du processus de vérification sur la représentation des états en mémoire. Nous aborderons les méthodes de compression d'états dont notamment la méthode des Δ-*markings* (3.3.3).

3.3.1 Le cas général

Dans le cas général, l'aspect réparti de l'exploration ne pose aucun problème pour la représentation des états. A priori, la représentation d'un état n'est pas sensée être dépendante du nœud sur lequel il se trouve. Ceci n'est toutefois pas le cas pour un certain nombre de technique de compression d'états qui visent à limiter la taille des états stockés en mémoire ou pour la technique des Δ-*markings* utilisée dans Helena et qui vise à ne pas représenter tous les états de façon explicite.

Nous présentons ici les modifications à apporter pour essayer d'adapter ces techniques de représentation ou de compression d'état tout en essayant de conserver leur efficacité. Nous verrons que dans le cas du Δ-*marking*, l'efficacité en terme de gestion de la mémoire ne peut se faire sans un compromis sur l'efficacité en terme de communications et donc de temps d'exécution.

3.3.2 Les techniques de compression d'état

Le cas des techniques de compression d'états est légèrement particulier. Une partie des techniques de compression d'état (comme par exemple la méthode de *state collapsing*) utilisent des structures de données additionnelles pour représenter les états.

Pour présenter la méthode de *state collapsing* brièvement, considérons deux états s_1 et s_2. Reprenons l"état s_1 précedemment soit défini comme suit :

$$s_1 : \{p_1, < (0, 1, [1, 3, 4] >\} + \{p_2, < 0, 2, [3, 5] >\}$$

et s_2 défini par :

$$s_2 : \{p_1, < (0, 1, [1, 3, 4] >\} + \{p_3, < 0, 3, [3, 5] >\}$$

On remarque que de nombreuses valeurs restent inchangées entre s_1 et s_2. Le marquage sur la place p_1 notamment, est rigoureusement identique pour les deux états. Pour éviter de dupliquer ces valeurs dans la mémoire, la méthode du *state collapsing* propose de ne stocker ce marquage d'une seule fois et de lui attribuer un identifiant codé sur quelques bits. Ainsi, lorsque l'état s_1 est

stocké, plutôt que de stocker le marquage de p_1 directement, c'est l'identifiant de marquage qui est codé tandis que le marquage associé est placé dans une table séparée.

Pour chaque place du réseau, on utilise alors une table dans laquelle seront stockés les différents marquages recontrés. Cette stratégie peut ensuite être étendue au différents domaines de couleurs du réseau. Dans notre exemple, le marquage de la place p_2 et celui de la place p_3 sont assez similaires. Pour éviter de dupliquer le tableau de ces marquages, on peut alors stocker ce tableau dans la table de la couleur associée et ne stocker que l'identifiant de ce tableau pour les états s_1 et s_2.

Cette technique permet donc de réutiliser des marquages sans les stocker de façon redondante. On aboutit alors à des gains mémoire importants. Cependant, les tables associant valeurs et identifiants ne sont remplies qu'à la volée en fonction des états explorés. Dans notre environnement réparti, les tables ne seront donc plus cohérentes sur les différents nœuds puisqu'elles dépendent directement du l'exploration et plus exactement du parcours suivi pour l'exploration. Ce parcours n'étant pas le même en séquentiel et en parallèle, les tables ne seront donc pas identiques.

Pour que la représentation des états reste homogène sur les différents nœuds du réseau, une première solution consiste à maintenir, sur chaque site, un réplicat de ces tables de hachages. Maintenir ces réplicats semble malgré tout une opération extrêmement coûteuse et complexe.

Pour éviter d'avoir à maintenir ces informations, nous avons décidé de ne pas représenter les états de façon homogène sur les différents nœuds du réseau. On permet alors à chaque nœud d'avoir sa méthode de représentation d'un état. Pour permettre malgré tout aux différents sites de pouvoir s'échanger correctement des états, la représentation des états utilisée dans les communications est effectuée de façon non compressée et se révèle donc homogène pour chaque nœud du réseau. La contrepartie consiste en des messages qui seront de taille plus importante puisque les états échangés ne sont plus compressés.

3.3.3 Les Δ-markings

Comme nous venons de le voir, l'aspect réparti du processus de vérification n'a qu'un faible impact sur les méthodes de compression d'états les plus courantes. Nous allons désormais nous pencher sur une méthode de représentation d'état semi-symbolique utilisée par Helena [111]. Cette méthode permet des représentations extrêmement efficaces et peut tout à fait se combiner avec un grand nombre de techniques de compressions d'états.

Nous commencerons donc par présenter brièvement cette méthode appelée méthode des Δ-*markings* (ou Δ-marquages) puis nous présenterons les solutions apportées pour adapter cette technique de représentation à un environnement réparti.

a. Présentation

La relation de transition des réseaux de Petri est un mécanisme déterministe. Le franchissement d'une transition en un marquage m mène à un unique marquage m'. En se basant sur ce déterminisme nous proposons de stocker certains marquages de l'ensemble d'accessibilité de manière non explicite, en représentant ces marquages sous la forme d'un pointeur sur un prédecesseur du marquage couplé avec l'instance de transition qui, franchie à partir du prédecesseur, donne le marquage en question. En raison du déterminisme de la relation de transition, cette représentation est non ambigüe bien qu'elle ne soit pas canonique puisqu'un marquage peut avoir plusieurs prédécesseurs. Les marquages représentés de cette manière dans l'espace d'état sont appelés Δ-marquages. Les marquages stockés de la manière usuelle dans l'espace d'état sont dits stockés explicitement, et les Δ-marquages sont dits stockés symboliquement.
Stocker un pointeur sur un marquage et une instance de transition au lieu du marquage complet devrait naturellement mener à une représentation plus compacte de l'espace d'état, spécialement

quand le vecteur d'état du système est grand. Néanmoins cette représentation a un désavantage : le test qui consiste à vérifier si un marquage m n'a pas encore exploré peut être significativement plus lent. Ce test implique généralement de comparer m à un (ou plusieurs) marquage(s) m' stocké(s) dans l'espace d'état. Dans le schéma de stockage classique, m' est codé par un vecteur de bits, et m est aussi codé de cette manière avant son insertion dans l'espace d'état. La comparaison peut alors être effectuée efficacement par une comparaison de vecteur à vecteur. Quand l'ensemble d'accessibilité contient des Δ-marquages, cette opération devient plus complexe. Supposons que nous avons une séquence de marquages $m_1, m_2, \ldots, m_n = m'$ telle que m_1 est stocké explicitement et chaque $m_i \neq m_1$ est un Δ-marquage qui pointe sur m_{i-1} avec l'instance (t_{i-1}, c_{i-1}) telle que $m_{i-1}[(t_{i-1}, c_{i-1})\rangle m_i$. Le principe est alors de remonter jusqu'à m_1 et de lui appliquer la séquence de franchissement $(t_1, c_1).(t_2, c_2)\ldots(t_{n-1}, c_{n-1})$ pour retrouver la "vraie" valeur de m'. Une fois cette opération réalisée, la comparaison de m et m' devient triviale.
Nous utiliserons le terme de *reconstitution* pour parler d'un tel processus, et la séquence franchie qui permet de reconstituer le marquage sera appelée une *séquence reconstituante*. Le principe du mécanisme de reconstitution peut être illustré à l'aide de la figure 3.12. Supposons, par exemple, que nous avons à reconstituer le marquage m. Pour cela nous devrons d'abord remonter jusqu'à m'. Puisque celui-ci n'est pas stocké explicitement, nous devrons encore remonter jusqu'à m_0 et lui appliquer finalement la séquence reconstituante $(t', c').(t, c)$. Cette opération nous permet de retrouver la valeur réelle du marquage m.

Le surcoût en temps introduit par notre méthode dépend directement de la lenteur du processus de reconstitution et donc des longueurs des séquences reconstituantes. Afin de placer une borne supérieur sur la longueur de ces séquences, nous reprenons l'idée de Geldenhuys [33] sur laquelle est basée sa stratégie de cache stratifié. Nous utilisons un paramètre k_δ, défini par l'utilisateur, qui appartient à l'ensemble des entiers naturels positifs. Durant l'exploration de l'espace d'état, chaque marquage rencontré à une profondeur d telle que $d \mod k_\delta = 0$ est stocké explicitement. Tous les autres marquages sont stockés symboliquement et pointent sur un de leurs prédécesseurs (par un arc en pointillé sur la figure). De cette manière, nous garantissons que la longueur de toute séquence reconstituante est bornée par $k_\delta - 1$. Cette idée est illustrée par la figure 3.12.

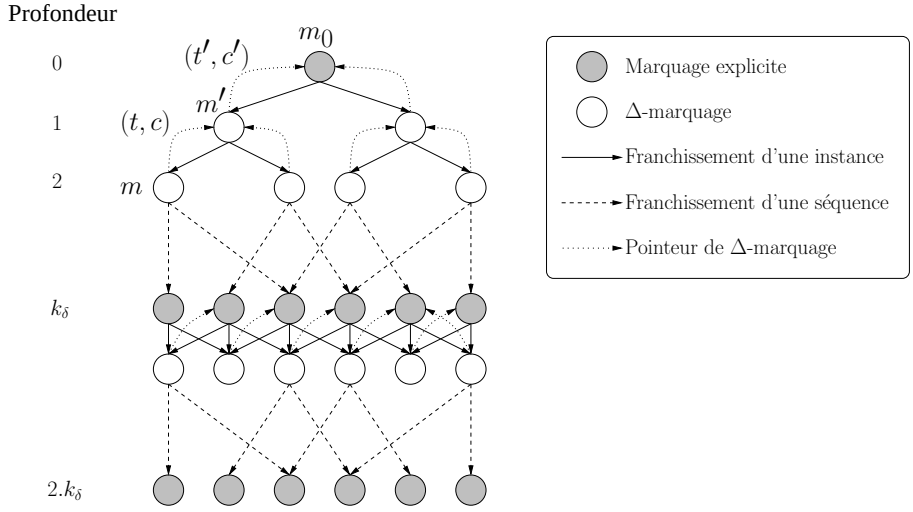


FIG. 3.12 – Un espace d'état avec des Δ-marquages

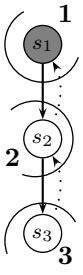
Plus de détails sur cette technique peuvent être trouvés dans [31] et [32].

b. Les Δ-markings en univers réparti

Lorsque nous avons essayé d'adapter cette méthode des *Δ-markings* à un environnement réparti, le principal problème qui s'est posé concernait la gestion des arcs arrières dans le représentation des état symbolique. Puisqu'un état symbolique est représenté à partir d'un de ces prédecesseurs, le problème de la représentation d'un état symbolique s_i dont le prédecesseur s_j n'appartient pas au même nœud que s_i se pose.

i. Une première solution pessimiste

Une première solution consiste à ajouter l'information nécessaire permettant à un état de retrouver le site sur lequel se trouve le prédecesseur à partir duquel il est représenté. L'information ainsi ajoutée n'est ainsi codée que sur $log_2(N)$ bits où N représente le nombre de nœuds mis en œuvre pour la vérification. Cet ajout de mémoire limité ne dégrade donc pas l'efficacité de la méthode.



La principale limitation provient du coût de reconstruction d'un état. Considérons par exemple le cas présenté à la figure ci-contre : s_1, s_2 et s_3 sont respectivement stockés sur les nœuds 1, 2 et 3. Si l'on cherche à reconstruire l'état s_3, il est nécessaire de remonter jusqu'à s_1 en passant par s_2. On pourrait imaginer un premier cas de figure où le site 3 envoie une requête à 2 qui lui-même envoie une requête à 1 qui répond à 2, etc ... Dans ce cas là, pas moins de 4 messages sont nécessaires pour reconstruire l'état s_3. Ce cas de figure correspond bien évidemment au pire cas pour cette solution : pour une séquence de reconstruction de taille s, $2s$ messages seront envoyés. Pour limiter le nombre de message, on peut alors opter pour une réponse directe de 1 à 3 en ne repassant pas par tous les intermédiaires lors de la redescente des réponses aux requêtes de reconstruction. Dans cette optique, on limite alors le nombre de communications à $s+1$ (cf. figure 3.14). Dans ce deuxième cas, le gain en terme de nombre de messages sera obtenu au détriment de la taille des messages puisque le nœud final devra recevoir tous les états intermédiaires pour pouvoir reconstruire le nœud final.

La figure 3.13 présente la structure d'une requête de reconstruction. Le premier élément représente le numéro du nœud qui a initié la requête ; cette valeur permettra au nœud possesseur du premier état explicite dans la séquence des prédécesseur de retrouver le nœud a qui envoyer la réponse. Le deuxième paramètre permet à l'initiateur de la requête d'identifier la requête. En effet, à chaque fois qu'un état nécessite d'être reconstitué, c'est qu'il devra ensuite être comparé à des états en attente d'être ajouté dans l'espace d'état. Cet identifiant permettra de retrouver tous les états qui doivent être comparés à cet état reconstitué. Le troisième paramètre représente l'adresse du prochain prédécesseur à chercher. Contrairement aux deux premiers paramètres qui ne sont jamais modifiés, cette valeur est modifiée par chaque nœud que la requête traverse. Les derniers éléments sont les couleurs et les identifiants des transitions à franchir. Notons également que lorsque l'état explicite a été trouvé, il est alors ajouter à la fin de cette liste.

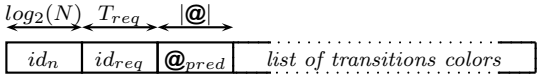


FIG. 3.13 – Structure d'une requête de reconstruction

Cette solution peut sembler lourde au premier abord étant donné le coût de reconstruction d'un message. Il est cependant interessant de noter un certain nombre d'avantages présentés par cette solution. Tout d'abord le gain mémoire obtenu suite à la parallélisation est le même que celui obtenu lors d'une exploration séquentielle – si l'on met de côtés les quelques bits ajoutés dans les états symboliques pour retrouver la bonne partition. De plus, la reconstruction d'un état ne s'effectue que lorsque l'on recherche si l'état n'est pas déjà présent dans l'espace d'état. Deux cas se présentent alors :

1. l'état a déjà été visité et se trouve par conséquent dans l'espace des états visités
2. l'état n'a pas encore été visité mais son ajout entraîne une collision dans la table de hachage de l'espace d'état

Le premier cas est le cas le plus difficile à limiter. Le second cas, quant à lui, est limité par l'aspect réparti du processus de vérification. En effet, pour une table de hachage de taille t et un espace d'états de taille N, la probabilité d'une collision est :

$$p_{séquentiel} = \frac{N}{t}$$

Dans un environnement réparti faisant intervenir n nœuds lors du processus de vérification, la probabilité devient alors :

$$p_{réparti} = \frac{N}{t \times n}$$

En augmentant la taille de la table de hachage on diminue donc les probabilités de collisions.
On peut alors supposer que le nombre de reconstructions d'état sera suffisemment faible pour composer leur coût. De plus, une optimisation simple consiste à repousser toute exploration d'un état ayant déclenché une demande de reconstruction. Ainsi, en attendant la reconstruction, une exploration à partir d'un autre peut alors débuter réduisant encore le coût d'une demande de reconstruction d'état.

Limiter les requêtes de reconstruction redondantes Dans certains cas, il est possible qu'une même requête soit envoyée plusieurs fois par un même nœud. Pour limiter les envois redondants, il est alors possible de gérer un cache d'états reconstitués. Cependant, il est nécessaire de traiter le cas particulier dans lequel une requête de reconstruction est émise alors que la même requête a déjà été envoyée par le même site et que la réponse n'a pas encore été reçue. Dans ce cas, l'état reconstitué n'est pas présent dans le cache, la requête sera alors ré-émise.
L'algorithme de construction est présenté à la figure 3.15. Lorsqu'un état s doit être ajouté à l'espace d'état, il doit être comparé à tous les états présents à la même entrée de la table de hachage. Lorsque certains de ces états doivent être reconstitués, on appelle la méthode Reconstruction_Request avec s et la liste des adresses des prédecesseurs à récupérer en paramètre. Pour chacune de ces adresses, on teste dans un premier temps si l'état associé n'est pas dans le cache (ligne 2). Si c'est le cas on peut alors directement comparer s à l'aide de l'état présent dans le cache. Si l'état reconstruit est l'état s (ligne 3) on peut alors conclure que s ne doit pas être ajouté à l'espace d'état. On libère alors toutes les éventuelles requêtes pour s (ligne 4) puis on peut sortir.

Dans le cas où l'état n'est pas en cache, il faut le reconstruire. La première étape consiste à regarder si la même requête n'a pas déjà été envoyée (ligne 8). Si c'est le cas, on ajoute s à la liste des états qui devront être comparé à un état reconstruit à l'aide de cette requête (ligne 10). Dans le cas contraire, on crée une nouvelle requête envoyée au nœud responsable du stockage de l'état stocké à l'adresse a et on associé s à cette requête (ligne 13 à 15).

A la reception d'une réponse on peut alors retrouver tous les états à comparer à des états reconstruits à l'aide de l'état reçu.

ii. Une seconde solution optimiste

Une seconde solution apportée à l'adaptation de la méthode des Δ-*markings* à un environnement réparti a été proposée. Cette méthode vise à limiter le coût des reconstruction. Elle se base sur une approche optimiste du comportement de la méthode des Δ-*markings* en environnement réparti en faisant l'hypothèse que le partitionnement des états aboutira à un degré de localité élevé.

L'idée est donc ici de ne pas générer de communications lors de la reconstruction d'un état. Pour cela, il est nécessaire que toute l'information requise pour la reconstruction soit disponible sur chaque nœud. Le principe est donc le suivant : chaque état de tranfert est stocké de façon explicite.

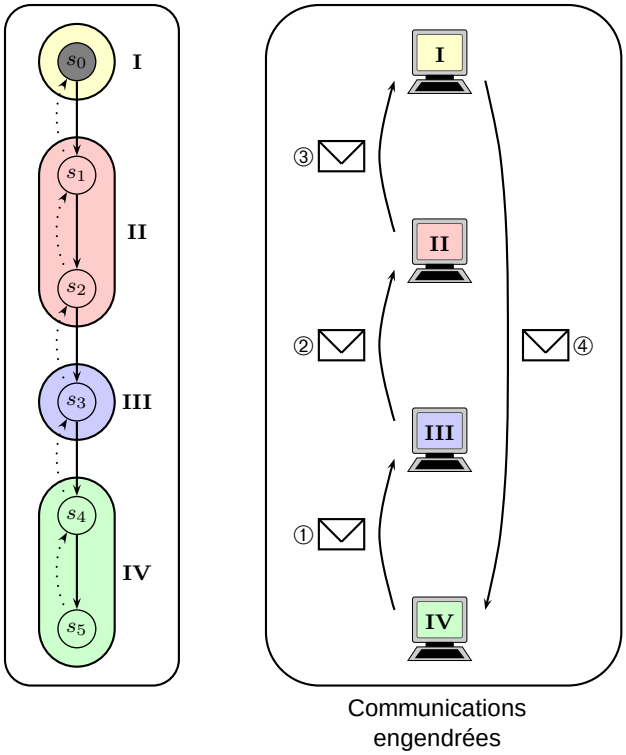


FIG. 3.14 – Exemple de communications pour une reconstruction pour les Δ-*markings* répartis pessimistes.

```
Reconstruction_Request (s, addresses)
1   for a ∈ addresses do
2     if a ∈ cache then
3       if compare(s, state(a)) then
4         free_request(s);
5         return true;
6       end if
7     else
8       if a has already been requested then
9         if s ∉ a_compare list then
10          add(s, a_compare list);
11        end if
12      else
13        req ← create_request(a, s);
14        send(req, owner(a));
15        store(req);
16      end if
17    end if
18  end for
19  return false;
```

FIG. 3.15 – Algorithme de reconstitution d'un Δ-*marking*

Lors de la reconstruction d'un état, le premier état stocké de façon explicite est donc forcément présent sur le site. Aucune communication n'est donc nécessaire.

Cette approche n'introduit donc aucune communication supplémentaire pour la reconstruction. Cependant, l'efficacité de la méthode s'en trouve réduite. Le gain obtenue en mémoire va donc fortement dépendre du degré de localité du partitionnement. Ici, la longueur maximale des séquences de reconstruction ne dépend plus seulement du paramètre k_δ mais également des longueurs des séquences d'exploration locales.

Optimisation Pour limiter la dégradation de l'efficacité de cette approche pessimiste nous autorisons – dans certains cas – la duplication de Δ-marquages. En utilisant des mécanismes d'anticipation (*children look-ahead*), on peut alors essayer d'évaluer la pertinence d'une duplication d'état symbolique voire explicite.

Un premier cas de figure est présenté à la figure 3.16. Ici, 3 états s_0, s_1 et s_2, tous stockés symboliquement dans la méthode séquentielle, se répartissent sur deux sites P_1 et P_2 dans la version répartie. Si l'on utilise l'approche pessimiste, s_1 et s_2 sont alors stockés de façon explicite. En anticipant l'exploration de s_2, il est alors possible de dupliquer s_1 sur P_1, entraînant alors le stockage sous forme symbolique de s_1 et s_2 sur P_1. Cette optimisation garantit un véritable gain mémoire dans la majeure partie des modèles puisque l'espace mémoire nécessaire au stockage d'un état explicite est bien souvent largement supérieur à celui utilisé pour le stockage de deux états symboliques.

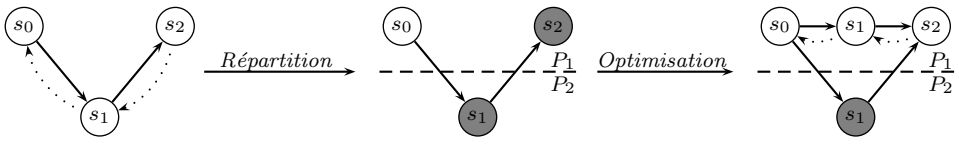


FIG. 3.16 – $1^{ère}$ optimisation de l'approche optimiste du Δ-marking réparti

Un second cas de figure est présenté à la figure 3.17. Dans la méthode séquentielle, s_1 est dé-

sormais stocké explicitement. Et s_1 possède plusieurs successeurs appartenants à P_1. L'approche pessimiste entraîne une véritable dégradation du gain mémoire puisque tous les successeurs de s_1 seront stockés explicitement sur P_1. De la même manière, en anticipant l'exploration de s_1, il est alors possible de dupliquer s_1 pour obtenir un stockage symbolique de tous ses successeurs.

Il est à noter que cette optimisation n'est bénéfique dans le cas où s_1 possède plus d'un successeur. Dans le cas contraire, la duplication de s_1 sur P_1 aboutit à un surcoût mémoire.

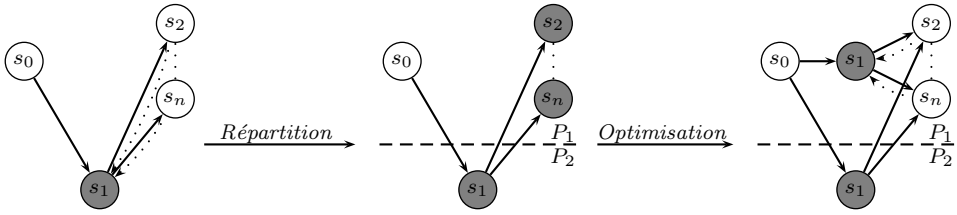


FIG. 3.17 – 2^{nde} optimisation de l'approche optimiste du Δ-marking réparti

Enfin, le dernier cas de figure à considérer est présenté à la figure 3.18. Dans ce cas là, s_2 est stocké de façon explicite dans la méthode séquentielle. La duplication de s_1 ne peut donc aboutir à un gain mémoire puisque s_2 sera de toute façon stocké explicitement sur P_1.

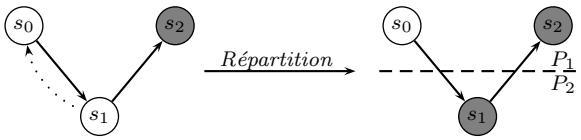


FIG. 3.18 – $3^{ème}$ cas : optimisation impossible de l'approche optimiste

iii. Evaluations et conclusion

Les résultats obtenus en utilisant l'algorithme optimiste pour les Δ-*markings* répartis sont présentés sur la figure 3.19. Selon le type de partitionnement effectué, il apparaît clairement que cette méthode n'offre pas du tout les mêmes résultats. Si l'on met en relation ces résultats avec ceux obtenus pour le degré de localité pour chaque type de partitionnement, on peut nettement voir que les deux valeurs sont fortemement corrélées.

Ici, l'utilisation d'une fonction de partition ayant pour but une répartition uniforme des états aboutit à une inefficacité quasi totale de cette adaptation des Δ-*markings*. Dès que l'on utilise des stratégies de partitionnemnent structurel, l'efficacité de cette approche s'en voit très nettement améliorée.
On peut enfin noter que l'utilisation des optimisations permet de récupérer une bonne partie de l'efficacité dans le cas de partitionnement peu efficace en terme de degré de localité. Notons cependant que dans ce cas, un certain nombre d'états sont dupliqués sur les nœuds. Même si cela aboutit à un gain mémoire comparé à l'utilisation de cette stratégie sans l'optimisation, la quantité de mémoire nécessaire reste généralement supérieure à celle nécessaire en séquentiel.

Remarque Dans certains cas où la méthode des Δ-*markings* optimiste fonctionne bien, on peut aboutir à un nombre de marquages stockés symboliquement supérieur à celui obtenu en environnement séquentiel. Ceci n'est pas dû à un *bug* mais découle du parcours du graphe d'accessibilité.

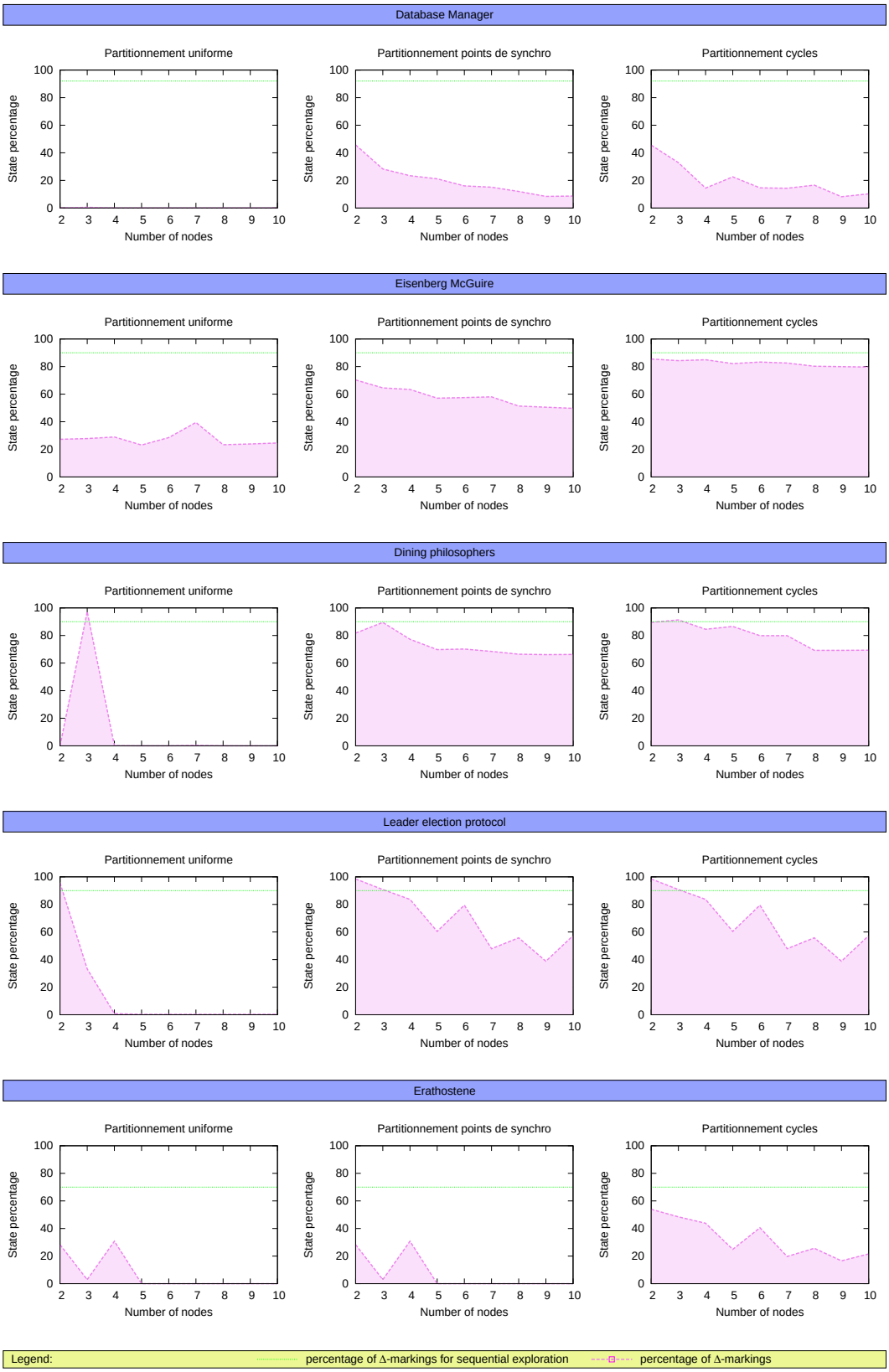


FIG. 3.19 – Nombre de Δ-marquages obtenus en réparti

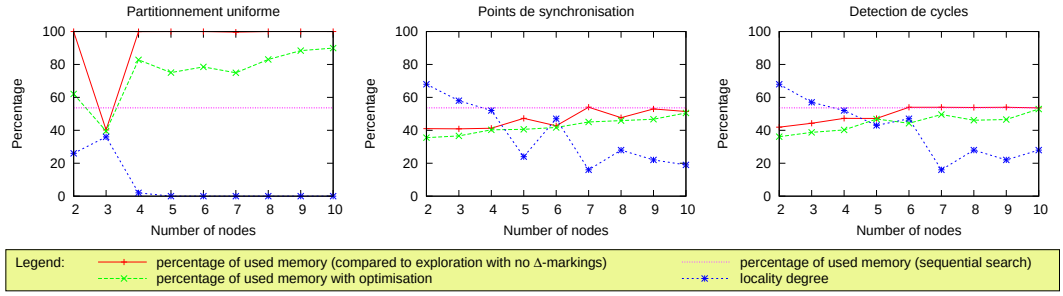


FIG. 3.20 – Impact des optimisations sur la mémoire (*Dining Philosophers*)

Considérons les deux parcours présenté à la figure 3.21. Dans le cas du parcours séquentiel, la séquence $s1 \dots s_i$ est explorée avant la séquence $s_n \dots sj$. Ainsi, lorsque s_j est visité pour la première fois, il est considéré comme étant à une profondeur $k_\delta - 1$, ses 3 successeurs seront donc stockés explicitement.

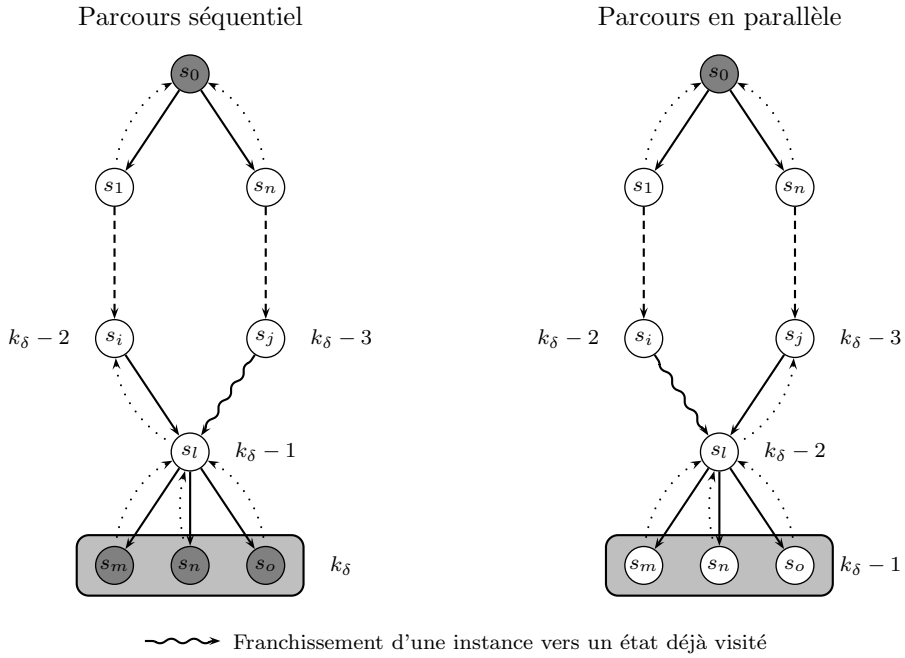


FIG. 3.21 – Impact du parcours global

Dans le cas d'une exploration parallèle, il est tout à fait possible que s_j soit visité en premier à la suite de la séquence $s_n \dots s_j$. Dans ce cas là, s_l est considéré comme étant de profondeur $k_\delta - 2$. Ses trois successeurs seront alors stockés symboliquement. La méthode des Δ-*marking* est donc dépendante du parcours global.

3.4 Rapport d'erreur

Lorsque l'on effectue de la vérification de propritété, il est essentiel de pouvoir rapporter une séquence d'exécution invalidant ladite propriété dans le cas où un telle séquence serait trouvée.

Dans l'approche séquentielle, obtenir cette séquence est extrêmement facile. La pile de recherche représentant à tout instant une exécution possible du système considéré, il suffit de traverser cette pile pour reconstituer la séquence fautive.

Lors d'une recherche en parallèle, il est inpensable de maintenir une pile d'exploration globalement cohérente. La recherche étant répartie sur chaque nœud, elle ne suit plus l'ordre de la recherche séquentielle. Chaque nœud maintient donc une pile de recherche locale, maintenir une pile globale se révèle donc impossible.

3.4.1 Etat de l'art

La construction du rapport d'erreur dans un environnement à mémoire répartie a été abordée dans [80]. Les auteurs y présentent une méthode simple qui consiste à permettre à chaque nœud de pouvoir reconstruire la pile d'exploration globale à tout instant.

Le principe est le suivant : pour chaque état envoyé, la pile de parcours est envoyée afin de pouvoir à tout moment reconstituer la séquence issue de l'état initial dans sa totalité.

3.4.2 Une autre approche

La précédente solution est assez coûteuse en terme de mémoire. Il est alors intéressant pour nous de noter que cette stratégie peut être contournée dans le cas où les Δ-*markings* sont utilisés. En effet, dans le cas des Δ-*markings* (que ce soit avec l'approche optimiste ou pessimiste), chaque état stocké symboliquement est représenté à l'aide d'un "pointeur" vers un de ses prédecesseur. On peut alors reconstruire une partie de la pile de parcours. Le seul problème concerne les états stockés explicitement qui eux, ne possèdent aucun lien vers un prédecesseur.

Une solution simple et très efficace en coût mémoire consiste à ajouter à chaque état explicite les mêmes informations que celles utilisées pour représenter un état symbolique.

Lorsque l'on cherche à reconstruire la trace, il suffit alors de remonter la série de liens jusqu'à l'état initial tout en reconstituant les états symboliques. Cette méthode présente l'énorme avantage de limiter la taille des messages puisque la pile n'est pas envoyée avec chaque état. Elle fait en fait partie de la représentation des états.

3.5 Indéterminisme

Le fait que l'exploration du graphe d'accessibilité s'effectue de façon désordonnée lors d'une exploration répartie pose la question de l'impact de l'indéterminisme sur les différentes méthodes présentées.

Partitionnement L'indéterminisme introduit lors de l'exploration répartie du graphe d'accessibilité ne peut avoir aucun impact sur le partitionnement lui-même puisqu'il ne dépend pas de l'exploration. Ici, quelque soit l'ordre d'exploration le partitionnement restera le même.

Les Δ-*markings* L'indéterminisme peut également avoir un impact sur l'efficacité de l'adaptation des Δ-*markings* optimistes. Il est en effet possible d'obtenir un nombre de Δ-*markings* différents pour plusieurs exécutions. Cependant, dans la pratique, la différence est assez faible. Prenons par exemple le cas du dîner des philosophes dont le nombre d'arcs est plus de 1à fois supérieur au nombre d'états, dans ce modèle le nombre de revisite d'état est très élevé et l'indéterminisme peut alors avoir un impact important. On observe cependant, dans la pratique une très faible variation du nombre de Δ-*markings* pour diverses exécutions. Considérons par exemple le cas du dîner des philosophes. Nous avons effectué 10 exécutions successives pour 5 nœuds et 10 nœuds. Le nombre d'états symboliques obtenu pour chaque exécution est présenté dans le tableau ci-dessous :

n	1	2	3	4	5
5	3 805 496	3 806 098	3 805 546	3 804 731	3 805 486
10	3 310 336	3 311 539	3 309 802	3 321 364	3 313 993

n	6	7	8	9	10
5	3 805 608	3 805 635	3 805 414	3 805 215	3 805 598
10	3 311 123	3 312 477	3 318 545	3 316 125	3 311 952

Le nombre d'états total étant de 4 766 585, on oberve donc une variation du nombre de Δ-*markings* extrêmement faible entre chaque exécution pour cinq nœuds. Les variations sont alors un tout petit peu plus élevée lorsque l'on utilise 10 nœuds mais restent extrêmement faibles (moins de 1%).

Rapport d'erreur Lors de la vérification de propriétés à la volée, dès qu'un état invalidant la propriété est détecté, l'exploration peut s'arrêter et reporte la séquence d'actions invalidant la propriété. L'indéterminisme joue alors ici un rôle qui peut être important. Le fait que l'exploration soit désordonnée implique que plusieurs exécutions successives peuvent aboutir à la détection d'états invalides différents et donc de séquences différentes et surtout de longueurs potentiellement très variables. L'idéal étant d'obtenir une séquence la plus petite possible cet indéterminisme peut poser problème.
Dans le cas d'une exploration effectuée séquentiellement deux approches sont possibles pour détecter des séquences invalides les plus petites possibles. La première consiste à utiliser un parcours en largeur. On sait alors que le premier état invalide détecté sera celui contenu sur la séquence la plus petite. Cependant, l'utilisation d'un parcours en largeur diminue généralement l'efficacité des techniques de réductions d'ordre partiel – que nous aborderons dans le chapitre **??**- et peut alors aboutir malgré tout à la détection de séquences invalides de longue taille.
Dans un contexte réparti, cette solution n'est de toute façon plus envisageable puisque le caractère désordonné du parcours fait que le premier état invalide détecté n'est plus forcément l'état appartenant à la séquence invalide la plus courte.

Une seconde solution pour détecter les séquences les plus courtes consiste à lancer plusieurs exécutions successives en limitant la profondeur maximale d'exploration. A chaque exécution, on réduit la profondeur maximale jusqu'à obtenir la séquence minimale. Contrairement à la première approche, cette méthode reste utilisable en réparti sans être impactée par l'indéterminisme.

3.6 Terminaison

Comme nous l'avons expliqué dans le chapitre 2, la détection de la terminaison des algorithmes répartis a fait l'objet de nombreuses contributions. Notre approche repose sur la détection de l'inactivité de tous les nœuds esclaves. L'algorithme que nous utilisons pour la détection de la terminaison est une adaptation de l'algorithme présenté dans [125] (cf. b.). Cette détection est déléguée au *manager*. Un nœud est inactif lorsqu'il n'a plus d'états à explorer et qu'il n'a plus de messages entrants ou sortants en attente d'émission ou de réception. Lorsqu'un nœud passe dans l'état incatif, il envoie un message au *manager*. Lorsque ce dernier a détecté que tous les nœuds sont inactifs, il doit ensuite vérifier qu'aucun message n'est actuellement transmis sur le réseau. Pour cela, lorsqu'un nœud envoie un message au *manager* pour lui signifier son inactivité, il envoie par la même occasion le nombre de messages qu'il a envoyé et le nombre de messages reçus. La terminaison est donc détectée lorsque tous les nœuds sont inactifs et que le nombre de messages reçus est égal au nombre de messages envoyés. Une fois la terminaison détectée, le *manager* réveille tous les nœuds esclaves en diffusant un message de terminaison.

3.7 Equilibrage de charge

Plusieurs algorithmes d'équilibrage de charge pour le model checking réparti on été présentées dans [81, 78, 79]. La plupart des approches reposent sur un partitionnement plus "fin" de l'espace d'état. Plutôt que d'associer une partition à un nœud, on découpe ici l'espace d'état en un nombre de partitions supérieur au nombre de nœuds. Un nœud est alors responsable de plusieurs partitions. Lors d'un rééquilibrage, l'attribution des partitions est modifiée pour prendre en comptre le déséquilibre de la charge.

Dans tous les cas, le rééquilibrage est déclenché lorsqu'un nœud voit sa quantité de mémoire passer sous un certain seuil. Différentes stratégies d'exploration sont alors proposées sans réelles expérimentations.

Lorsqu'un rééquilibrage est décidé, deux stratégies sont alors possibles. Considérons la configuration présentée à la figure 3.22. Supposons que la mémoire disponible sur le nœud n_1 soit passée sous le

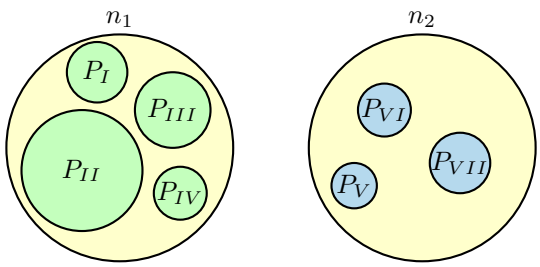


FIG. 3.22 – Exemple de partitionnement

seuil autorisé. Une réorganisation des partitions est alors décidée par le *manager*. Supposons alors que les *manager* décide de déléguer la partition P_{III} à n_2 ; deux stratégies sont possibles :

1. le nœud n_1 envoie tous les états de la partition P_{III} à n_2 puis libère la mémoire associée à cette partition
2. le nœud n_1 libère la mémoire associée à cette partition sans envoyer les états à n_2

On remarque ici que le coût des deux opérations n'est pas du tout le même. Dans le premier cas, le nombre de communications peut alors être considérablement augmenté. Dans le second cas, en revanche, le rééquilibrage ne rajoute pas *directement* de communications. Cependant, il est alors possible que tous les états de P_{III} soient revisités. Cette exploration redondante génère alors de nouvelles communications en plus d'augmenter de façon potentiellement importante le temps d'exécution.
Dans Cyclades, nous n'avons pas implémenté de mécanismes d'équilibrage de charge. Les raisons de ce choix proviennent de l'incompatibilité du rééquilibrage avec un certain nombre de techniques de réductions et/ou de compressions d'états utilisées dans Cyclades et dans la majeure partie des outils de vérification. Nous allons d'ailleurs aborder rapidement le cas du Δ-*marking*. Nous reviendrons sur le cas des *stubborn sets* dans le chapitre 6. Nous proposerons également une autre solution pour l'équilibrage de charge dans le chapitre 5.

Remarque sur le Δ-marking Comme nous venons de le voir, le rééquilibrage de charge consiste à réorganiser les partitions sur l'ensemble de nœuds du réseau. La technique des Δ-*markings* étant basé sur la notion de prédecesseur et plus exactement d'adresse du prédecesseur, ce changement de localisation des états pose problème.
Considérons alors le cas de l'algorithme optimiste. Une adresse vers un prédecesseur est constitué d'une adresse dans l'espace d'état couplé avec une adresse vers le nœud sur lequel est stocké cet état. Ici, si l'état change de nœud, l'adresse n'est alors plus valide. Une modification simple consiste donc

à ne pas couplé l'adresse dans l'espace d'état avec lenœud sur lequel l'état est stocké mais avec la partition à laquelle appartient l'état. Cette modification légère est facile à apporter à l'algorithme. Elle implique toutefois une légère surconsommation mémoire puisque le nombre de bits nécessaires à l'encodage du nœud sera forcément plus petit que le nombre de bits nécessaire à l'encodage de la partition.
Cette simple modification n'est toutefois pas suffisante pour adapté l'algorithme optimiste aux stratégies d'équilibrage de charge. En effet, lors d'une réorganisation, un état s stocké à une adresse a_s sur un nœud peut entrer en collision avec un état s' stocké à la même adresse sur le nœud sur lequel s est déplacé. Là encore, une solution assez simple peut être envisagée. Elle consiste à ne plus stocker tous les états sur un même nœud par le biais d'une unique table de hachage mais d'utiliser *plusieurs* tables de hachages et donc plusieurs zones de stockage. L'idée est donc d'avoir une zone de stockage d'état par partition. Les collisions ne sont alors plus possibles lors d'une réorganisation. Le prix à payer est une nouvelle légère surconsommation mémoire dûe à la multiplication des diverses structures des zones mémoire. Ce surcoût reste toutefois assez faible (une dizaine de Mo environ dans Cyclades).
L'adaptation de l'algorihtme pessimiste est quant à elle beaucoup plus complexe et semble difficilement envisageable sans modifications profondes et/ou assez coûteuses ou inefficaces. En effet, dans cette approche nous avons interdit la possibilité de représenter un état par un prédecesseur n'étant pas sur le même nœud. Une première possibilité serait de n'autoriser la représentation d'un Δ-*marking* qu'à partir d'un prédecesseur issu de la même partition. Or, puisque le nombre de partition augmente lorsque l'on veut faire de l'équilibrage de charge, le degré de localité risque de baisser de façon significative, dégradant alors l'efficacité de la méthode. L'autre solution serait alors de basculer sur la méthode optimiste. Ceci étant difficilement envisageable.
Notons enfin que dans tous les cas, si l'on choisit de ne pas envoyer tous les états lors d'une réorganisation, l'utilisation des Δ-*markings* n'est alors plus possible.

3.8 Conclusion

Dans ce chapitre, nous avons présenté les mécanismes du model checking parallèle dans un environnement à mémoire répartie. Nous nous sommes attachés à essayer de conserver les méthodes déjà existantes et à les adapter de façon la plus efficace possible à ce nouvel environnement.

Les résultats obtenus montrent qu'il est possible de conserver l'efficacité de ces techniques. Toutefois, le compromis entre efficacité de la méthode et efficacité de la répartition est parfois inévitable. Le cas le plus évident concerne l'adaptation de la méthode des Δ-*markings*. La stratégie optimiste nous permet de ne pas rajouter de communications lors de l'exploration parallèle.

Toutefois, l'efficacité des Δ-*markings* ne peut être garantie sans une fonction de partition permettant un fort degré de localité. Pour conserver toute l'efficacité des Δ-*markings*, il convient alors de permettre qu'un état soit représenté à partir d'un prédecesseur n'appartenant pas à la même partition. Cette stratégie, peut alors s'avérer coûteuse en terme de communications surtout si, là encore, le degré de localité du partitionnement n'est pas suffisemment élevé.
Ces résultats mettent donc en évidence l'importance de la fonction de partition, notamment en terme de localité.

Au delà des aspects d'efficacité des diverses techniques de réductions, on peut également s'intéresser à l'efficacité de la répartition en terme de temps d'exploration. La figure 3.23 présente justement les temps d'exploration obtenus pour 4 modèles de la distribution. Nous avons utilisé ici la méthode des Δ-*markings* optimistes couplés à une fonction de partition basée sur la détection de cycle. La parallélisation a donc bien permis de réduire le temps d'exploration. La réduction du temps d'exploration est continue et tend à s'attenuer à partir d'un certain seuil où le nombre de nœuds utilisés devient trop important comparé aux nombre d'états explorés.

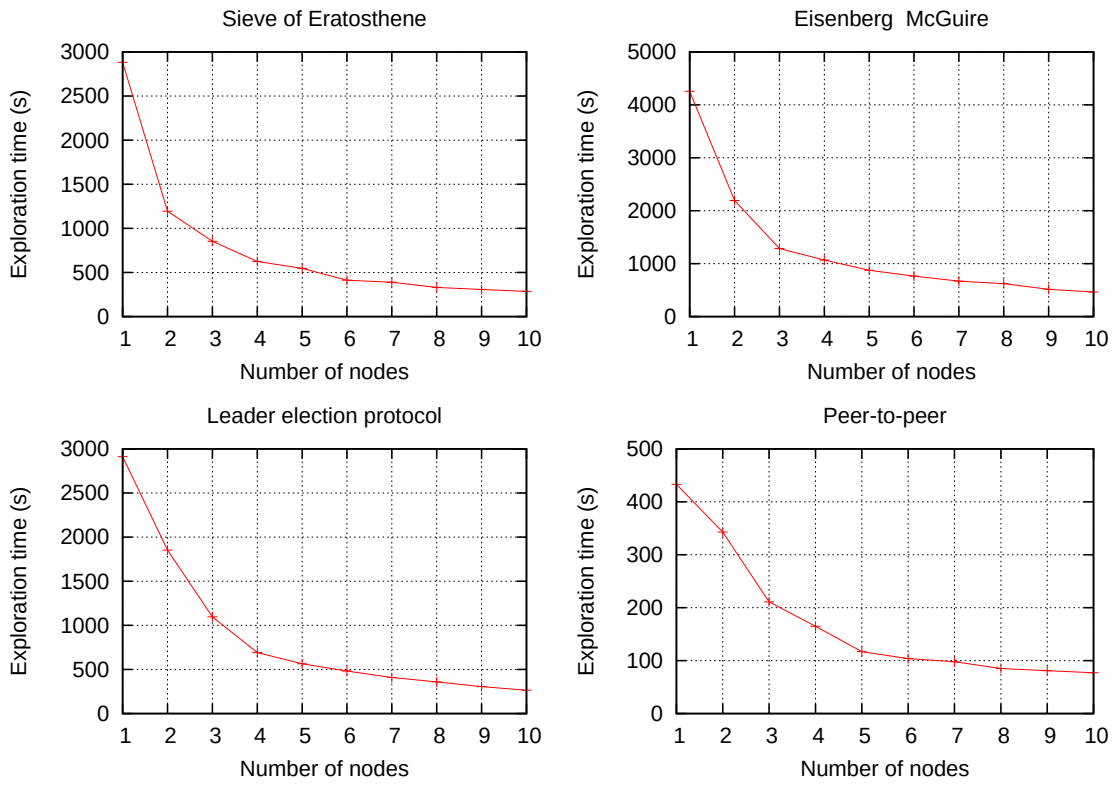


FIG. 3.23 – Evaluation du temps d'exploration pour le model checking réparti avec Cyclades.

Une première perspective de travaux futurs concerne la vérification de propriétés LTL. La vérification de ce type de propriétés a déjà fait l'objet de nombreux travaux intéressants [91, 92, 93, 94, 95] que nous souhaitons intégrer à Cyclades.

Chapitre

4

Model checking parallèle

Sommaire

Depuis de nombreuses années, le CPU (*Central Processing Unit*, ou microprocesseur) fait l'objet d'une course à la performance, mettant en scène des concurrents qui se livrent une lutte toujours plus acharnée pour l'obtention de la première place dans les benchmarks. Dans cette optique, les différents constructeurs adoptent souvent des approches différentes, parfois même divergentes. Par contre il reste une donnée constante : pour une architecture donnée, la voie principale permettant d'améliorer les performances est l'augmentation de la fréquence. Malheureusement, cette méthode a plusieurs inconvénients. Premièrement, elle est de plus en plus difficilement applicable car l'augmentation de la fréquence n'est pas proportionnelle à l'évolution des procédés de fabrication. Deuxièmement, ladite évolution devient difficile et coûte cher. Enfin troisièmement, la consommation de courant - donc la dissipation thermique - augmente bien plus vite que la fréquence pour une finesse de gravure donnée. En d'autres termes, les processeurs évoluent moins vite et ont tendance à chauffer de plus en plus.

Pour faire face à ces divers problèmes, les constructeurs se sont donc tournés vers les architectures

multi-cœur[1] et multi-processeurs. Le principe étant alors d'utiliser en parallèle l'équivalent de deux processeurs cadencé à une certaine fréquence f plutôt que d'essayer de développer un monoprocesseur cadencé à une fréquence de $2f$, beaucoup plus difficile à mettre en œuvre.

Cette nouvelle orientation des constructeurs relance donc une partie de la recherche dans le domaine des systèmes parallèles à mémoire partagée. En effet, pour tirer profit de ces architectures, il est nécessaire d'adapter les programmes à ce nouvel environnement et de modifier – si c'est possible – les algorithmes en conséquences.

Le model checking n'échappe pas à cette nouvelle donne qui encourage les chercheurs à ce pencher sérieusement sur le model checking parallèle dans les environnements à mémoire partagée. Ce chapitre présente donc les résultats des premières recherches dans cette voie et notre apport à cet aspect du model checking.

Certaines techniques visant à combattre l'explosion pouvant être assez gourmandes en terme de calculs (Δ-*markings*, compression d'état, réductions d'ordre partiel), il est alors intéressant d'essayer de réduire le temps d'exploration en parallélisant ces calculs.

4.1 Présentation et comparaison au model checking réparti

Dans le model checking parallèle, comme dans le model checking réparti, l'exploration de l'espace d'état se fait de manière simultanée par plusieurs processus. La principale différence réside dans le partitionnement de l'espace d'état. Dans le cas du model checking réparti, l'espace d'état doit être partitionné au départ ce qui n'est pas le cas du model checking parallèle. En effet, la mémoire – et donc l'espace d'état – étant partagée, il n'y a pas de risque d'explorer des états de façon redondante. On peut alors gérer beaucoup plus simplement l'exploration et garantir un certain degré de localité pour chaque processus.

Nous verrons également qu'un équilibrage de charge simple peut être beaucoup plus aisément instauré qu'en réparti.

Une dernière différence concerne l'accès à l'espace d'état. Dans un environnement à mémoire partagée, l'accès à l'espace nécessite de prendre un certain nombre de précaution du fait du caractère concurrent des accès mémoires inhérent à ce type d'environnement.

La principale similitude concerne le parcours de l'espace d'état. Là encore, il n'y a plus aucune cohérence globale du parcours. La cohérence ne peut être garantie que localement à chaque processus.

4.2 Etat de l'art

Peu de travaux ont abordés le cas du model checking parallèle jusqu'à présent. Les premiers ont été proposés dans [86, 88] puis plus récemment par Holzmann [89, 90]. Dans ces derniers papiers, l'auteur y présente un algorithme parallèle pour le model checker Spin. Le principe de base est bien évidemment de permettre le partage et l'exploration concurrente de l'espace d'état par plusieurs threads.

[1] un processeur multi-cœur désigne un processeur composé de plusieurs cœur (ou unité de calculs) gravé au sein de la même puce. De façon simpliste, on peut définir un processeurs multi-cœur comme un système multiprocesseur dans lequel les différents processeurs sont réunis en un seul processeur. Nous ne ferons pas dans cette thèse de distinction entre ces deux types de systèmes et, par abus de langage, nous désignerons comme système mutli-processeur, tant les systèmes multi-processeur que les systèmes multi-cœur.

Comme nous l'avons précisé précédemment, le partitionnement de l'espace d'état ne peut souffrir du problème de duplication d'état. La répartition des états est donc beaucoup plus souple qu'en réparti puisqu'elle ne peut jamais mener à des explorations redondantes de l'espace d'état. Dans [90], chaque état effectue un parcours en profondeur à partir d'un état jusqu'à une certaine profondeur p. Lorsque la profondeur p est atteinte, l'exploration des états suivants est déléguée à un autre thread qui va lui aussi effectuer une exploration jusqu'à une profondeur p. Le résultat final est donc une exploration effectuée par strates de hauteur p (cf. figure 4.1).

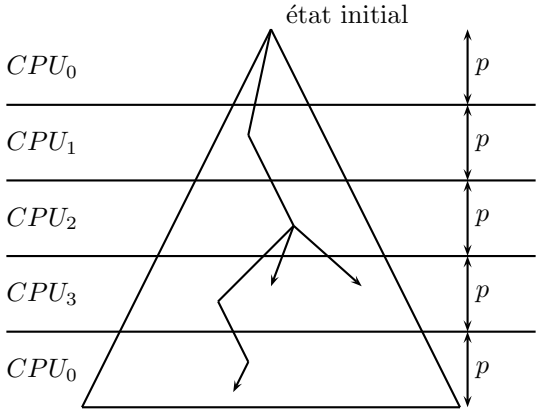


FIG. 4.1 – Exploration par strate de l'espace d'état.

Notons bien ici que l'exploration elle-même n'est pas effectuée par strate mais ce sont uniquement les différentes séquences d'exploration qui sont découpées en séquence de profondeur bornée. Cette approche permet de favoriser le degré de localité de l'exploration tout en permettant de paralléliser l'exploration de l'espace d'état sur les différents *threads*.

4.3 Architecture générale

L'architecture de la version parallèle de Cyclades est présentée à la figure 4.2. Chaque *thread* peut accéder de façon concurrente à l'espace d'état.
Cependant, le parcours global n'étant plus cohérent, il n'est pas possible d'utiliser une seule et unique pile de parcours ; chaque *thread* possède alors sa propre pile de parcours locale.

Pour communiquer entre eux, les *threads* utilisent des files de messages. Ces files doivent donc être protégées par des verrous. Pour éviter tout risque de goulot d'étranglement dû à un grand nombre d'accès concurrents, nous n'avons pas utilisé une seule et unique file de communication partagée par tous les *threads*.
Chaque *thread* possède une unique file de communication en entrée et une en sortie. Les files ne sont donc partagées que par 2 *threads*. Il n'est en effet pas nécessaire qu'un *thread* puisse communiquer avec tous les autres *threads*. Contrairement au model checking réparti où les nœuds sont responsables d'une partition, ici, les *threads* n'ont pas d'états attribués spécifiquement. Lorsqu'un *thread* veut transmettre à état à un *thread*, cela peut être n'importe lequel des *threads* c'est pourquoi nous avons opté pour cette architecture qui présente l'avantage de limiter les données partagées et les accès concurrents à ces données.

4.4 Gestion de l'espace d'état

L'idée de base de notre approche repose bien évidemment sur le partage de l'espace d'état. En effet, il serait tout à fait possible d'exécuter plusieurs processus de vérification sur une machine multi-

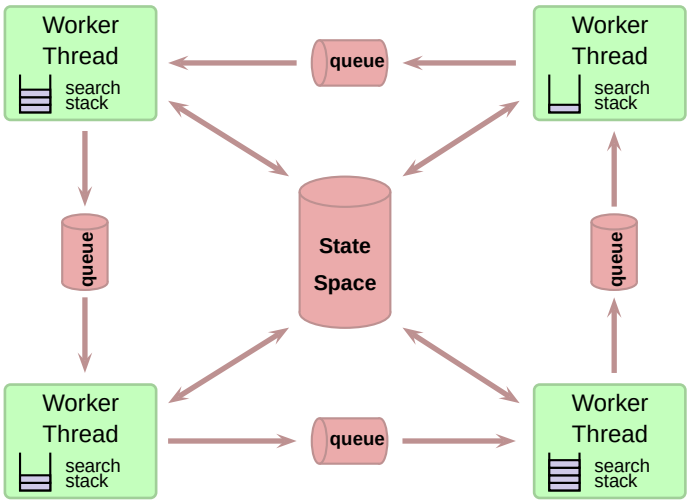


FIG. 4.2 – Architecture de la version parallèle de Cyclades (4 *threads*).

processeurs sans partager l'espace d'état. Cette solution reviendrait tout simplement à considérer qu'une machine composée de N processeurs peut être vue comme N nœuds sur le réseau. Cette solution ne demanderait aucune modification du code. Mais outre la lourdeur de cette solution, le problème provient du partitionnement. En effet, si l'on utilise chaque processeur comme un nœud, le nombre de partitions augmente de façon impressionnante. Cette multiplication des partitions nuit forcément à la vérification puisqu'elle risque de limiter grandement l'efficacité de certaines méthodes – comme les Δ-*markings* et les réductions d'ordre partiel. Quelle que soit l'approche considérée, plusieurs threads vont accéder de façon concurrente à l'espace d'état.

4.4.1 Exploration de l'espace d'état

L'exploration de l'espace d'état s'effectue de façon très similaire à celle adoptée par le model checking réparti. La seule différence se situe lors de la décision de stopper l'exploration pour la déléguer à un processus tiers. Dans le cas du model checking réparti, un test est effectué pour chaque état : un nœud teste s'il est responsable de la partition à laquelle cet état appartient. Si c'est le cas il continue son exploration, sinon il transmet l'état au nœud concerné (cf. 4.2).

Ici, un état n'*appartient* pas à un processus particulier. Lorsqu'un processus a atteint une certaine profondeur de recherche, il transmet les état suivants à un autre processus. Dans le cas contraire, il continue son exécution de façon tout à fait normale. L'algorithme utilisé est présenté à la figure 4.3 où i est le numéro du *thread* courant et T le nombre total de *threads*. Il est très similaire à celui utilisé en séquentiel (fig. 3.1). Cependant, à chaque appel, la profondeur du parcours est ajoutée. C'est cette profondeur de parcours qu'il faut tester à chaque fois pour savoir si le *thread* doit ou non déléguer l'exploration de l'état courant. S'il doit déléguer l'exploration, il le place alors dans sa file sortante pour le passer au *thread* suivant (ligne 2). La valeur de K_DEPTH est une constante spécifiée par l'utilisateur.

Equilibrage de charge En limitant la taille des files de communications entre les *threads*, on peut effectuer un équilibrage de charge simple. Lorsqu'un *thread* veut déposer un état s dans la file de son voisin, si la file est pleine, plutôt que d'attendre de pouvoir y déposer son message, on peut

```
DFS (s, local_depth)
1   if local_depth = K_DEPTH then
2      enqueue(queue_{(i+1)%T}, s);
3   else
4      if s ∉ Visited then
5         Visited ← Visited ∪ {s}
6         stack ← stack ∪ {s}
7         for s' ∈ Successor(s) do
8            DFS(s', local_depth + 1)
9         end for
10        stack ← stack \ {s}
11     end if
12  end if
```

FIG. 4.3 – Procédure DFS en environnement à mémoire partagée

autoriser le *thread* à poursuivre son exploration à partir de s. Une file de communication est pleine signifie en effet qu'un *thread* est occupé et donc non inactif.

4.4.2 Accès l'espace d'états

Comme nous l'avons déjà précisé, le stockage de l'espace d'état utilise une table de hachage. Pour chaque état, on calcule la clé correspondante pour la table de hachage. Pour chaque entrée de la table, plusieurs états peuvent être stockés. Les accès fréquents à cette table rendent inenvisageable la protection de cette ressource critique par un seul et même verrou. Nous utilisons alors un verrou par entrée dans la table de hachage. Lorsqu'un *thread* veut accéder à une entrée, il doit posséder le verrou correspondant.

Les risques d'attente des verrous sont assez limités puisque si la fonction de hachage est efficace, il n'y aura que très peu de collisions et la majeure partie des accès concurrents sera due à la recherche d'un état déjà visité. La taille de la table de hachage étant généralement assez grande (1 048 576 entrées par défaut dans Helena), les attentes de verrou seront normalement assez limitées.

Ici, le fait que le nombre d'états soit fini nous garantit l'impossibilité de famine d'un ou plusieurs *threads*.

Surcoût mémoire L'ajout de verrous pour protéger l'espace d'état entraîne un surcoût en terme d'utilisation de la mémoire. Ce coût dépend de la taille de la table de hachage et ne dépend pas du nombre d'états stockés. Dans le cas de Cyclades, la taille de la table de hachage est de 1 048 576 entrées par défaut. Si l'on considère qu'un verrou est représenté sur 24 octets, on obtient un surcoût mémoire d'un peu plus de 24Mo. Cette valeur représente la valeur initiale. Lors de l'exécution, cette valeur peut augmenter de quelques octets puisque chaque mutex conserve la liste des *threads* en attente du verrou. Ces quelques octets sont toutefois négligeables par rapport aux 24Mo.

Gestion des arcs arrières Contrairement à la parallélisation en environnement réparti qui nécessitait de gérer de façon particulière les liens vers les prédécesseurs, ici aucun travail particulier n'est a effectué. Pour faciliter l'utilisation des Δ-*markings*, il suffit de choisir une profondeur d'exploration p pour chaque *thread* telle que $p = k.\delta$ où δ et la profondeur utilisée pour la méthode des Δ-*markings* et k est un entier tel que $k > 0$. Dans ce cas, chaque état passé d'un *thread* à un autre est un état explicite.

4.5 Représentation et compression d'états

A priori, le multithreading ne pose pas de problème pour ce qui touche à la représentation d'un état. Ceci est vrai dans le cas général mais pas forcément lorsque l'on utilise certaines techniques de compression d'état.

En effet, les techniques de *bitstate hashing*, *state collapsing* ou encore de Δ-*marking*– toutes utilisables dans Helena– nécessitent de mettre en place des mécanismes de protection plus ou moins importants lorsque l'on se place dans un contexte multithreadé. L'utilisation, dans ces méthodes, de données partagées nécessite un certain nombre de précautions.

4.5.1 Exemple de la technique "*state collapsing*"

La méthode de *state collapsing* est une méthode de compression d'état particulièrement utilisée dans le model checking explicite. Son principe est assez simple : il consiste à utiliser des tables de taille fixe dans lesquelles on associe aux différentes valeurs possibles d'une couleur un identifiant codé sur quelques bits. La représentation d'un état ne se fait donc plus à l'aide des vraies valeurs des couleurs mais uniquement par le biais de ces identifiants.

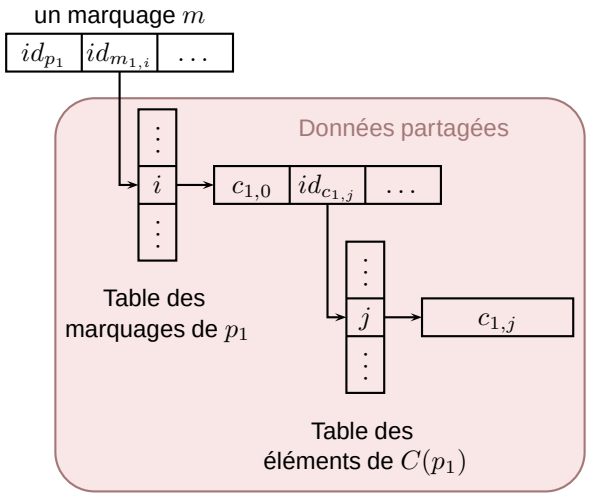


FIG. 4.4 – Principe du *state collapsing*

On peut ensuite généraliser cette méthode aux marquages de chaque place. Pour chaque place, une table associant une valeur de jeton à un identifiant unique peut être utilisée. Un état est alors représenté par une liste d'identifiant de jetons pour chaque places. Ces jetons sont alors représentés par des listes d'identifiants de couleurs. Une telle représentation est présentée à la figure 4.4.

Contrairement à l'espace d'état, les probabilité d'accès concurrents à des même entrées d'une table est beaucoup plus élevée pour le *state collapsing*. En effet, à chaque ajout d'état dans l'espace d'état ou à chaque lecture d'un état dans l'espace d'état, il est nécessaire d'accéder à ces tables. Pour chaque état, plusieurs tables sont accédées. Autant le décodage d'un état est effectué assez rapidement puisqu'il suffit de récupérer les identifiant pour accéder aux bonnes valeurs, autant l'encodage d'un état est plus coûteux : il faut en effet parcourir chaque table considérée à la recherche de la valeur à encoder. Si cette valeur est déjà présente, on peut alors en récupérer l'identifiant, sinon, il est nécessaire de la rajouter à la table et de lui allouer un nouvel identifiant.

Le nombre important d'accès à ces tables nous oblige à utiliser des mécanismes de verrouillage les plus fins possible. Il serait en effet impossible de verrouiller chaque table par un seul verrou. Là encore, on utilise un verrou par entrée dans chacune des tables.

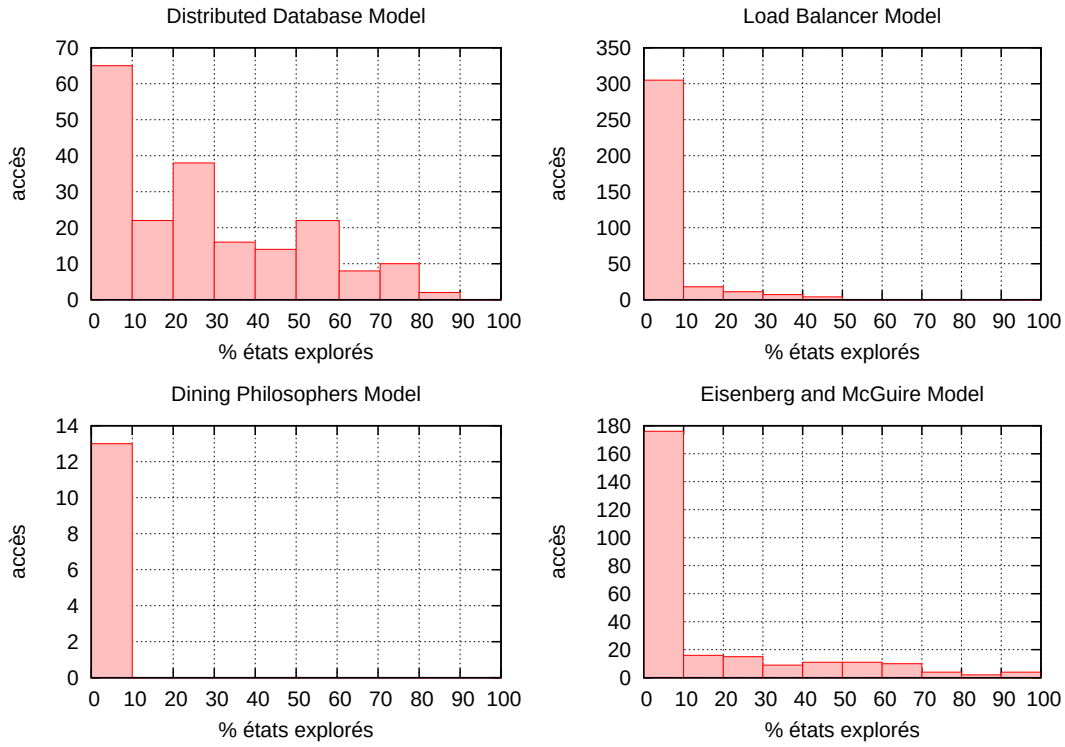


FIG. 4.5 – Nombre d'accès en écriture aux tables de *state collapsing* en fonction de l'évolution de l'exploration de l'espace d'états.

De plus, le nombre d'accès à ces tables en écriture n'est pas constant. Après une certaine phase de remplissage des tables, beaucoup de valeurs sont stockées et les accès sont essentiellement des accès en lecture. La figure 4.5 présente le nombre d'accès en écriture dans les tables en fonction de l'avancement du processus. On remarque, dans tous les cas, que la grande majorité des accès en écriture s'effectuent au tout début de l'exploration. Après cette première phase de remplissage, les accès sont essentiellement des accès en lecture. C'est pourquoi l'utilisation du modèle de coopération lecteur/rédacteur (p. 57) est plus approprié.

L'utilisation de ce modèle va pénaliser les écritures qui seront effectuées au début de l'exploration, puis va permettre à plusieurs *threads* d'accéder de façon concurrente aux différentes tables en lecture.

Surcoût mémoire Contrairement au surcoût mémoire pour la table de hachage de l'espace d'état, ici le surcoût mémoire n'est pas fixe et dépend du modèle. En effet, le nombre de verrous va dépendre du nombre de tables de compression. Ce nombre de tables dépend lui-même du nombre de places du réseau ainsi que du nombre de domaines de couleur. Le nombre de verrous dépend enfin de la taille réservée à chacune des tables. Notons toutefois qu'un verrou de type lecter/rédacteur nécessite plus de mémoire qu'un verrou simple comme utilisé pour la table de hachage de l'espace d'état.

4.5.2 Cas du Δ-*marking*

Comme nous l'avons vu dans le chapitre précédent, la technique du Δ-*marking* permet de représenter certains états à partir d'un prédécesseur. Pour reconstruire un tel marquage, il est nécessaire de parcourir l'espace d'état pour retrouver le prédécesseur et remonter la séquence des prédécesseurs jusqu'au premier état stocké explicitement.

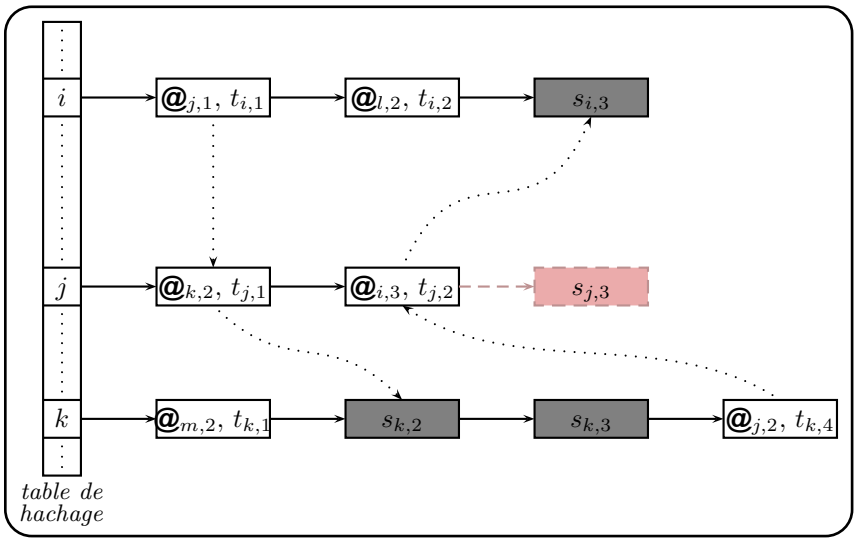


FIG. 4.6 – Exemple d'espace d'état utilisant la technique des Δ-*markings*.

Considérons alors le cas présenté sur la figure 4.6. Considérons alors que deux *threads* t_j et t_k sont en train de parcourir l'espace d'état. Supposons que t_j cherche à ajouter l'état $s_{j,3}$ qui sera stocké en troisième position à l'entrée j de la table de hachage. Supposons que, dans le même temps, le *thread* t_k cherche à ajouter un état à l'entrée k de la table de hachage.

Avant de pouvoir éventuellement ajouter les états souhaités, t_j et t_k doivent reconstituer tous les états stockés respectivement aux entrées j et k de la table de hachage. Pour ce faire, il se doivent de prendre le verrou sur ces deux entrées afin d'éviter tout risque de lecture ou écriture incohérente.

Cependant, lorsque t_k cherche à reconstituer le dernier état de l'entrée k, il se trouve alors dans l'obligation d'accéder à l'entrée j de l'espace d'état pour y récupérer le prédécesseur. Or, lorsque t_j à chercher à reconstituer le premier état de l'entrée j, il a, lui aussi, dû essayer d'accéder à l'entrée k pour y récupérer le prédécesseur. On se retrouve alors dans une situation où t_j possède le verrou de l'entrée j et attend celui de l'entrée k, et où t_k possède celui de l'entrée k et attend celui de l'entrée j. Cette situation aboutit donc à un *deadlock*.

Pour remédier à ce problème, il est nécessaire d'autoriser l'accès en lecture à plusieurs *threads* simultanément et interdire à un *thread* de posséder plus d'un verrou exclusif en écriture sur la table de hachage. L'utilisation du modèle de synchronisation lecteur/rédacteur est donc impératif. Lorsqu'un *thread* cherche à ajouter un état à l'espace d'état, il commence par prendre le verrou en lecture associé à l'entrée correspondante. Puis il vérifie que l'état qu'il souhaite ajouter n'est pas déjà présent. Si l'état est déjà visité, le verrou peut alors être relâché. S'il n'est pas déjà visité, le *thread* relâche malgré tout son verrou en lecture puis cherche à l'acquérir en écriture. Une fois le verrou acquis, l'état peut alors être ajouté à l'espace d'état sans aucun risque d'interblocage puisque le *thread* n'a plus besoin d'accéder à d'autres entrées de la table de hachage.

Cas particulier Lorsqu'un *thread* cherche à ajouter un état dans l'espace d'état, un comportement erroné peut arriver une fois que le *thread* a obtenu le verrou en écriture. Supposons qu'un *thread* t_1 veuille ajouter un état s_i à une entrée i, il commence par parcourir tous les états présents dans cette entrée. Si s_i n'est pas déjà présent, t_1 relâche le verrou en lecture puis cherche à l'acquérir en écriture. Dans le même temps, il est alors possible qu'un *thread* t_2 ait cherché à ajouter un état s'_i à la même entrée. Si l'on suppose que t_2 obtient le verrou en écriture avant t_1, lorsque t_1 l'obtient, l'espace d'état à changé. Selon les implémentations, dans le pire des cas, t_1 peut alors écraser l'état s'_i ou, dans le meilleur des cas, stocker s_i à la suite de s'_i ce qui peut aboutir à une duplication d'état si $s_i = s'_i$.

```
state_space_add (s)
1   local pos_start ← 1;
2   local key ← hash_key(s);
3   while true do
4
5     // step 1. look for state s
6     read_lock(hash_i);
7     for i from pos_start to size(state_space[key]) do
8       if s = reconstitute(state_space[key][i]) then
9         read_unlock(hash_i);
10        return false;
11      end if
12    end for
13    pos_start ← size(state_space[key]);
14    read_unlock(hash_i);
15
16    // step 2. add state s
17    write_lock(hash_i);
18    if pos_start = size(state_space[key]) then
19      add(s);
20      return true;
21    end if
22    write_unlock(hash_i);
23  end while
```

FIG. 4.7 – Ajout d'un Δ-*marking* dans l'espace d'état.

Pour éviter tout risque de comportement incorrect, il est alors nécessaire qu'un *thread* obtenant un verrou en écriture vérifie que l'espace d'état n'a pas changé entre le moment où il a rendu son verrou en lecture pour acquérir celui en écriture. Pour cela, il faut comparer le nombre d'états stockés avant et après avoir obtenu le verrou en écriture (notons qu'aucun n'est ne peut être retiré de l'espace d'état). Si l'on détecte une modification, il faut alors ré-effectuer un certain nombre de tests pour éviter la duplication d'état. Si des états ont été ajoutés, ils l'ont été en fin de liste, c'est-à-dire *après* les états que l'on a déjà reconstitués pour les comparer à l'état à ajouter. Dans ce cas là, on peut alors se limiter à la reconstitution de ces nouveaux états.

L'algorithme d'ajout d'un état pour la technique de Δ-*markings* est présenté à la figure 4.7. Comme nous venons de l'expliquer, il est composé de 2 parties : une première qui consiste à vérifier si l'état s est déjà présent (lignes 5 à 14), puis une seconde dans laquelle on ajoute le nouvel état (lignes 16 à 22). Dans le cas où l'espace d'état à été modifié pour l'entrée considérée entre les deux étapes (ligne 18), on retourne à l'étape 1 pour effectuer les tests complémentaires sur les états ajoutés pendant l'intervalle de temps entre le relâchement du verrou en lecture et l'acquisition du verrou en écriture.

Surcoût mémoire Lorsque l'on utilise la technique des Δ-*markings*, il devient nécessaire de remplacer les verrous simples adopter dans la solution générale pour protéger l'accès à l'espace d'état par des verrou de type lecteur/rédacteur. Cette solution induit donc un surcoût un peu plus important que celui de la solution générale. Si l'on considère qu'un verrou de type lecteur/rédacteur est codé sur 128 octets, le surcoût mémoire passe d'environ 24Mo à un peu plus de 131Mo. De la même façon que pour les verrous simples, cette valeur peut augmenter légèrement au cours de l'exécution. Ce surcoût reste cependant acceptable au regard de la quantité de mémoire disponible pour l'exploration.

4.6 Rapport d'erreur

Dans un environnement à mémoire partagée, le problème du rapport d'erreur est exactement le même qu'en réparti puisqu'il est généralement construit à l'aide de la pile de parcours qui n'est ici plus cohérente globalement.

Les réponses que nous pouvons apporter ici sont donc les mêmes que celles présentées dans le chapitre précédent.

4.7 Détection de la terminaison

Les conditions de terminaison de l'algorithme multithreadé sont similaires à celles de l'algorithme réparti. L'exploration est considérée comme terminée lorsque tous les processus sont inactifs et qu'il n'y a plus de messages en transit.
Pour détecter l'inactivité des *threads*, nous avons utilisé une simple variable globale (`global_idle`) qui sauvegarde le nombre de *threads* à l'état inactif. Cette valeur est incrémentée par chaque *thread* lorsque sa file de messages entrante est vide (ligne 2 de la fonction `check_termination`) puis est décrémentée lorsqu'un *thread* redevient actif (ligne 7 de la fonction `worker`). Cette variable globale est protégée par un verrou spécifique (`idle_lock`).

Le comportement des *threads* est présenté par la fonction `worker` ; tant que la terminaison n'a pas été détectée, chaque *thread* peut extraire un message de sa file entrante (ligne 12). Si le message reçu n'est pas un message de terminaison, une procédure d'exploration est lancée à partir de l'état récupéré.

Lorsqu'un *thread* dépose un message dans une file, si cette file était préalablement vide, il est alors possible que le *thread* correspondant soit à l'état inactif. Le *thread* doit alors envoyé un signal de réveil (ligne 5 de la fonction `enqueue`) pour réveiller le destinataire potentiellement en attente (ligne 5 fonction `worker`).

Lorsque la file entrante est vide, la fonction `check_termination` est appelée. C'est dans cette fonction que la variable globale `global_idle` est mise à jour et que la terminaison est détectée. Pour cela, si un *thread* t détecte que la variable `global_idle` est égale au nombre de *threads*, on peut supposer que l'exploration est terminée. Avant de pouvoir conclure, il faut toutefois vérifier que plus aucun message n'est présent dans un file de communications. Il est en effet tout à fait possible que le dernier *thread* t' à s'être mis en attente ait déposer un message dans la file de son successeur t'' puis l'ait réveillé avant de se placer à l'état inactif. Il est alors possible que t'' n'est pas eu le temps de mettre `global_idle` à jour alors que t détectait que la valeur de cette variable avait atteint le nombre de *threads*. Pour éviter toute fausse détection de la terminaison, lorsque t détecte que `global_idle` a atteint la valeur seuil, il doit encore parcourir toutes les files à la recherche d'une file non vide. S'il en trouve une (ligne 7 de la fonction `check_termination`, alors on peut conclure que l'exploration n'est pas terminée. Dans le cas contraire, un message de terminaison est placé dans chacune des files, puis chaque *thread* est réveillé (lignes 13 à 22).

```
main ()
1   global_idle ← 0;
2   for i from 1 to T do
3     Create_Thread(i, worker);
4   end for
5   enqueue(queue_0, s_0);
6   worker();

worker ()
1   while true do
2     mutex_lock(queue_id_lock);
3     if queue_id = ∅ then
4       if check_termination() = false then
5         cond_wait(queue_id_cond);
6         mutex_lock(idle_lock);
7         global_idle − −;
8         mutex_unlock(idle_lock);
9       end if
10    end if
11
12    s ← dequeue(queue_id);
13    mutex_unlock(queue_id_lock);
14    if s ≠ termination then
15      DFS(s, 0);
16    else
17      break;
18    end if
19  end while

enqueue (queue_i, s)
1   mutex_lock(queue_i_lock);
2   queue_i ← queue_i ∪ s;
3   if size(queue_i) = 1 then
4     // Wake up thread that may be waiting
5     cond_signal(queue_i_cond);
6   end if
7   mutex_unlock(queue_i_lock);

check_termination ()
1   mutex_lock(idle_lock);
2   all_idle ← (+ + global_idle = T);
3
4   if all_idle then
5     for i from 0 to T − id and all_idle do
6       mutex_lock(queue_i_lock);
7       if queue_i ≠ ∅ then
8         all_idle ← false;
9       end if
10      mutex_unlock(queue_i_lock);
11    end for
12
13    if all_idle then
14      for i from 0 to T − id do
15        mutex_lock(queue_i_lock);
16        enqueue(queue_i, termination);
17        mutex_unlock(queue_i_lock);
18        cond_signal(queue_i_cond);
19      end for
20      enqueue(queue_id, termination);
21    end if
22  end if
23  mutex_unlock(idle_lock);
24  return all_idle;
```

FIG. 4.8 – Procédure principale et procédure pour chaque *thread*

4.8 Evaluations

L'objectif principal de cette approche est la diminution du temps d'exécution. Nous avons effectué plusieurs expérimentations des algorithmes présentés dans ce chapitre à l'aide de notre outil Cyclades. Ces résultats sont présentés sur la figure 4.9. Ces résultats sont issus des tables B.15 et B.16.

Pour chaque modèle nous avons affiché le gain de temps obtenu en utilisant la méthode des Δ-*markings* seule dans un premier temps puis en utilisant les Δ-*markings* couplés à la technique du *state collapsing*. Nous avons également présenté le gain théorique. La formule du gain de temps est :

$$G = \frac{T_{seq}}{\tau_n}$$

où τ_n représente le temps obtenu en utilisant n *threads* et T_{seq} représente le temps d'exploration séquentiel. Le gain théorique correspond quant à lui au résultat de la formule du gain en utilisant comme valeur de τ_n, la valeur de $\frac{T_{seq}}{n}$, c'est à dire n.
Le résultats sont encourageants et mettent en évidence des gains de temps très intéressants. Dans la majeure partie des modèles étudiés, les gains de temps obtenus sont compris en 75 et 100% du gain théorique.

Dans le cas des modèles des philosophes et de l'anneau, le gain est quasiment optimal et même supérieur au gain théorique pour 2 et 3 *threads* dans le cas des philosophes. Ceci est du au nombre de Δ-*markings* obtenus et donc aux temps de reconstruction qui ne sont pas tout à fait les mêmes en parallèle ou en séquentiel (cf figure 3.21).

Le cas du modèle Eisenberg & McGuire Curieusement, le modèle *Eisenberg & McGuire* offre des performances assez faibles. L'explication est relativement simple. Si l'on se réfère à la courbe présentée à la figure 4.10, on observe un comportement extrêmement particulier. En effet, très rapidement, la taille de la pile devient extrêmement faible. Cette valeur passe même certaines fois sous la barre des 20 états qui est la valeur de la profondeur maximale d'exploration pour nos *threads*.
Cette taille de pile extrêmement faible aboutit donc forcément à une très mauvaise parallélisation puisque les *threads* ne pourront être correctement approvisionnés. Les temps d'inactivité sont alors importants, réduisant par là même le gain de temps de façon impressionnante.

Parallèlement à cela, on observe que le nombre de nouveaux états explorés atteint très rapidement son maximum. Très rapidement les états explorés sont uniquement des états déjà visités. Ceci engendre donc un comportement très particulier dans lequel les *threads* n'effectuent plus que des accès à l'espace d'état et aux files de messages sans aucune véritable exploration.

4.9 Conclusion

Dans ce chapitre, nous avons présenté plusieurs algorithmes permettant la parallélisation de l'exploration de l'espace d'état sur des machines multiprocesseurs. Les similitudes avec les problématiques de la parallélisation en environnement réparti existent mais malgré tout, le fait que les données soient ici partagées facilite l'adaptation de certaines méthodes.

La technique des Δ-*marking* s'adapte ici parfaitement sans aucune dégradation possible de la méthode si ce n'est le léger surcoût mémoire lié aux structures de données additionnelles utilisées pour la synchronisation.

Notons également que la question du partitionnement ne se pose absolument plus. Ici les états ne sont pas liés à un processus particulier mais peuvent être exploré par n'importe quel *thread*.

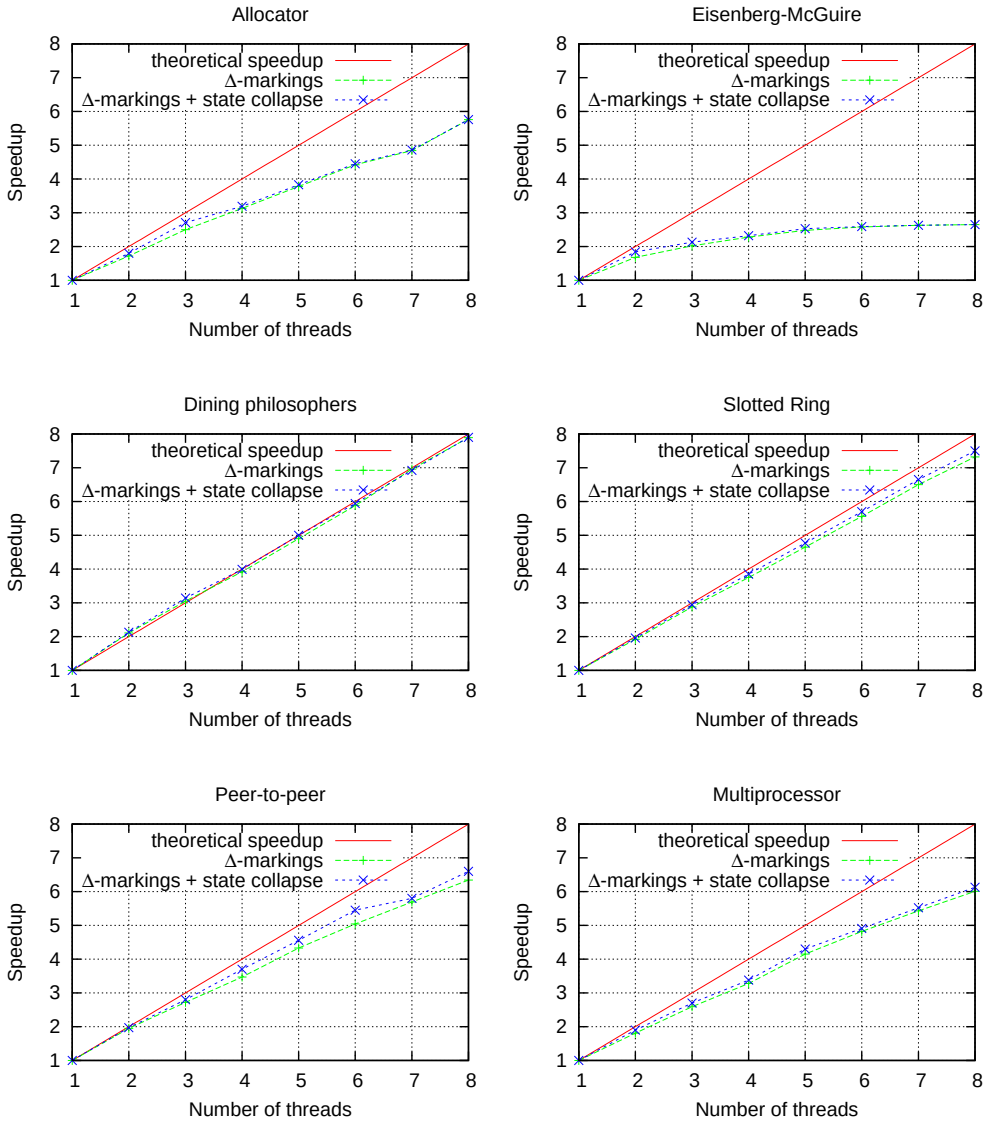


FIG. 4.9 – Temps d'exécution

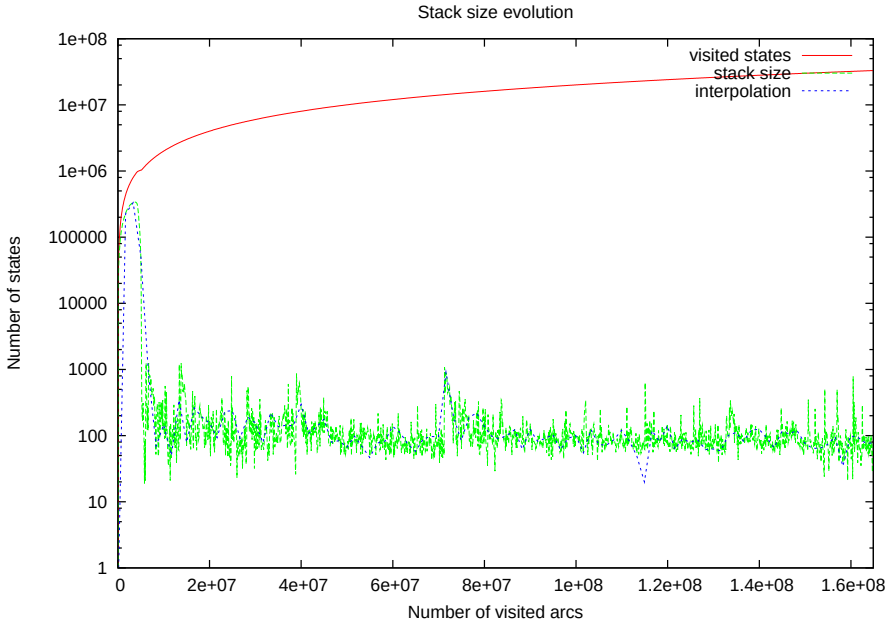


FIG. 4.10 – Evolution de la taille de la pile au cours de l'exploration pour le modèle *Eisenberg & McGuire* (échelle logarithmique).

Les nombreux verrous utilisés ne dégradent finalement pas l'efficacité de l'exploration et les gains de temps observés sont tous extrêmement bons. Et ce, même dans le cas d'utilisation de techniques qui accèdent fréquemment à des données partagées. Dans ces cas là, l'utilisation de modèles de synchronisation adaptés s'est révélée particulièrement importante.

Notons toutefois deux problèmes qui restent communs tant à la parallélisation en environnement réparti qu'en partagé : le rapport d'erreur et les techniques d'ordre partiel. Nous aborderons ce dernier point dans le chapitre 6. Le premier point quant à lui, nous oblige à conserver la même approche que celle présentée au chapitre précédent.

Là encore, une évolution à court terme consiste à adapter des méthodes de vérification de propriétés LTL dans ce nouvel environnement. Un algorithme assez efficace a déjà été présenté dans [87].

Chapitre

5 Model checking réparti multithreadé

Sommaire

Une fois les deux premières version de Cyclades développées, à savoir la version répartie et la version multithreadée, il nous a paru intéressant de développer une version couplant les deux approches dans l'optique d'obtenir les avantages des deux stratégies.

Cette approche permet également d'envisager des évolutions futures intéressantes que nous présenterons à la fin du chapitre. Celui-ci est organisé de la façon suivante : dans une première section, nous présenterons rapidement l'état de l'art et ses limitations puis nous présenterons les deux approches architecturales utilisées avant de présenter quelques résultats qui ne seront que préliminaires puisque l'environnement de test n'a pu nous permettre d'approfondir les expérimentations.

5.1 Etat de l'art : Eddy_Murφ

A notre connaissance, Eddy_Murφ [75] est le seul model checker réparti *et* multithreadé. Eddy_Murφ est la version multithreadée du model checker Murφ.

Ce model checker est en fait composé de 2 threads : l'un dédié à la gestion des communications, l'autre à l'exploration de l'espace d'état. Cette méthode permet donc à d'explorer l'espace d'état tout en traitant les envois et les réceptions de messages.

Le *thread* dédié aux communications gère aussi bien la gestion des messages entrants (déserialisation des états, ...) que la gestion des messages sortants (sérialisation des états, *bufferisation* des messages, ...).

La communication entre les deux *threads* s'effectue à l'aide de deux files de messages. L'une stockant les états reçus qui doivent donc être explorés. L'autre stockant les états à envoyer. Le *thread* gérant les communications effectue alors une attente active successivement sur les messages entrants puis sur les messages sortants.

5.1.1 Limitation

La principale limitation d'Eddy_Murφ réside dans le choix de n'utiliser que deux threads. Comme nous l'avons expliqué précédemment, les systèmes multiprocesseurs sont en plein développement et ne se limitent déjà plus à 2 cœurs. Des processeurs quadri et octo-core sont déjà disponibles. La solution proposée par Eddy_Murφ ne pourra donc pas tirer profit de ces évolutions.

5.2 Idée générale

Nous présentons donc ici des stratégies permettant d'utiliser les systèmes multiprocesseurs et ce, de manière à utiliser toutes les possibilités proposées par ces systèmes, quel que soit le nombre de processeurs.

Nous proposons ici deux approches différentes. L'une est une extension de la solution architecturale proposée par Eddy_Murφ : à savoir qu'un thread est totalement dédié aux communications, les autres sont dédiés à l'exploration.

La seconde approche consiste à autoriser chaque thread à explorer et communiquer ; plus aucun thread n'est alors dédié spécifiquement aux communications. Dans cette solution, tous les threads ont le même comportement.

Dans ces deux approches, les *threads* ne communiquent pas entre eux *pour l'exploration*. C'est l'une des principales différences avec la version présentée dans le chapitre 4. Dans nos approches, les *threads* effectuant l'exploration de l'espace d'état ne s'alimentent pas entre eux en terme de charge d'états. La seule source d'états provient des messages reçus. Une première conséquence de cette stratégie est la suppression des files de communications entre les *threads* explorateurs.

5.3 Première approche : extension d'Eddy_Murφ

La première approche que nous avons considérée consiste en une extension de l'architecture d'Eddy_Murφ. L'objectif étant de permettre à plusieurs threads d'effectuer une exploration concurrente de l'espace d'état. On ne pose alors aucune limitation sur le nombre de *threads* utilisables tout en gardant un *thread* totalement dédié à la gestion des communications. Cette architecture est présentée à la figure 5.1.

5.3.1 Gestion des communications

La gestion des communications est très fortement similaire à celle proposée dans Eddy_Murφ. Ici, l'accès aux primitives – et donc aux *buffers* – de communication n'est pas partagé puisque seul

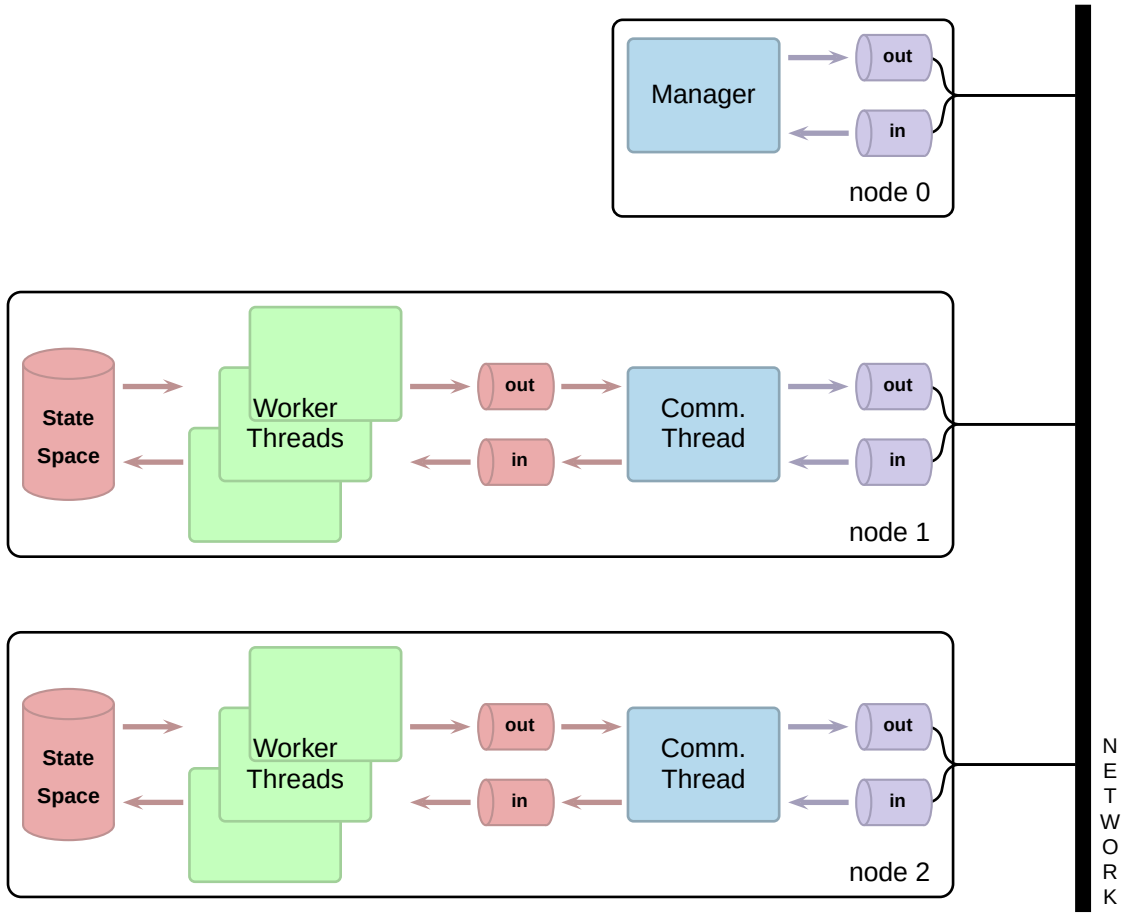


FIG. 5.1 – Architecture répartie multithreadée avec un *thread* dédié aux communications (4 threads par nœud).

le communicateur les utilise. Les états reçus sont alors placés dans une file d'attente partagée par tous les *threads*. Pour éviter des accès trop fréquents à cette file partagée, à chaque fois qu'un *thread* accède à la file, il ne récupère pas *un seul* état à partir duquel il pourra commencer une exploration mais il récupère directement un message dans son intégralité. La méthode de *bufferisation* des messages nous permet en effet de regrouper plusieurs états dans un seul et unique message.

Cette technique de *bufferisation* nous permet alors de limiter le goulot d'étranglement lié à l'utilisation d'une unique file partagée. Elle limite en effet le nombre d'acquisition de verrou et réduit donc le risque d'attente du verrou et les dégradations de performances qui pourraient en découler.

5.3.2 Détection de la terminaison

La détection de la terminaison fonctionne toujours de la même façon, à savoir : plus de messages en transit (ie. tous les messages envoyés ont été reçus) et chaque nœud est inactif.

Il est donc nécessaire d'ajouter un mécanisme permettant la détection de l'inactivité d'un nœud. Elle peut être détectée lorsque plus un seul *thread* explorateur n'est en train d'explorer, que plus aucun message ne doit être envoyé par le communicateur et que la file de communication interprocessus est vide.

Pour des raisons de simplicité, il semble plus intéressant de déléguer la détection de l'inactivité au *thread* communicateur. Deux conditions nécessaires de l'inactivité sont l'absence de messages entrants et sortants. Cette détection liminaire peut être aisément effectuée par le communicateur. Lorsque ces deux conditions sont réunies, le communicateur doit vérifier que tous les *threads* explorateurs sont inactifs et que la file de communication est vide. Ceci peut alors être effectué de façon tout à fait similaire à la solution utilisée dans le chapitre précédent, à savoir par le biais d'une variable partagée recensant le nombre de *threads* inactifs.

Lorsque l'inactivité est détectée par un site, celui peut alors envoyer les statistiques qui permettront au *manager* de détecter la terminaison de façon tout à fait similaire à la solution répartie classique.

A la réception d'un message de terminaison, le *thread* communicateur peut alors réveiller chaque *thread* explorateur et lui signaler la terminaison de l'algorithme.

5.3.3 Comportement des différents processus

Nous décrivons ici les comportements de deux types de processus : les exploreurs et le communicateur.

a. Les exploreurs

L'algorithme présenté à la figure 5.2 décrit le comportement global d'un *thread* exploreur. Tant que la terminaison de l'algorithme réparti n'a pas été détectée, l' explorateur effectue alternativement deux phases distinctes. La première (lignes 4 à 14) consiste à récupérer d'éventuels messages entrants dans la file de message partagées. Pour cela, le *thread* prend le verrou exclusif associé à cette file. Une file d'attente vide (test de la ligne 6), place le processus dans l'état *inactif*. Avant de se mettre dans cet état, il vide cependant ses *buffers* d'émission afin d'éviter une non-terminaison de l'algorithme. Une fois les *buffers* vidés (ligne 7), le *thread* peut alors signifier alors à tous les autres sont nouvel état en incrémentant la valeur de la variable globale `global_idle` qui sauvegarde le nombre de *threads* inactifs à un moment donné.

Une fois placé dans l'état inactif, le *thread* se met en attente d'un signal de réveil (ligne 9) qui, une fois reçu, signifie qu'un nouveau message est disponible dans la file.

```
explorer ()
1   msg ← ∅;
2   while not termination do
3
4      // Début de section critique
5      mutex_lock(incoming_mutex);
6      if incoming_queue = ∅ then
7         flush_all_buffers();
8         global_idle++;
9         cond_wait(incoming_mutex);
10        global_idle−−;
11     end if
12     msg ← dequeue(incoming_queue);
13     mutex_unlock(incoming_mutex);
14     // Fin de section critique
15
16     if msg = termination then
17        termination = true;
18     else
19        for each state s in msg do
20           DFS(s);
21        end for
22     end if
23  end while
```

FIG. 5.2 – Algorithme de l'explorer

```
communicator ()
1   while not termination do
2
3      // 1. Traitement des messages entrants
4      while incoming_message = true do
5         msg ← Receive();
6         mutex_lock(incoming_mutex);
7         incoming_queue ← incoming_queue ∪ msg;
8         if Size(incoming_queue) = 1 then
9            cond_signal(incoming_mutex);
10        end if
11        mutex_unlock(incoming_mutex);
12     end while
13
14     // 2. Traitement des messages sortants
15     mutex_lock(outgoing_mutex);
16     while outoing_queue ≠ ∅ do
17        msg ← Pop(outgoing_queue);
18        Send(msg);
19     end while
20     mutex_unlock(outgoing_mutex);
21
22
23  end while
```

FIG. 5.3 – Algorithme du communicator

```
flush_all_buffers ()
1   mutex_lock(outgoing_mutex);
2   for each node n do
3      flush_buffer(n);
4   end for
5   mutex_unlock(outgoing_mutex);
```

```
flush_buffer (dest)
1   msg ← send_buffer[dest];
2   enqueue(outgoing_queue, msg, dest);
```

FIG. 5.4 – Algorithmes de vidage des *buffers* d'émission

Le *thread* peut alors passer à la seconde phase de l'algorithme ; à savoir la lecture du message. Dans le cas d'un message de terminaison, le *thread* peut alors terminer son exécution. Dans le cas contraire, le message reçu est en fait un agrégat d'états à explorer. Pour chacun de ces états, le *thread* déclenche alors une recherche en profondeur (ligne 20).

b. Le communicateur

Le communicateur alterne également deux phases bien distinctes : l'écoute de messages entrants (lignes 3 à 12) puis l'envoi de messages (lignes 14 à 20). Pour éviter tout risque d'interblocage, il est nécessaire que l'attente de messages entrants se fasse de façon non bloquante. En effet, si l'on considère le cas où 2 nœuds n_1 et n_2 sont utilisés pour la vérification. Supposons que n_2 soit inactif et que sur n_1, le communicateur soit en attente de message entrant, si au moins un *thread* exploreur est toujours en cours d'exécution sur n_1 et qu'il souhaite envoyer un message, le communicateur de n_1 ne l'enverra jamais puisqu'il est en attente d'un message entrant qui n'arrivera jamais. Pour éviter ce cas de figure, il est nécessaire pour le communicateur d'effectuer une attente active pour les messages entrants de même que pour les messages sortants.

Dans le cas où un message entrant a été reçu alors que la pile était vide, il est alors nécessaire d'émettre un signal pour réveiller un éventuel *thread* en attente de message entrant (ligne 9 correspondant à la ligne 9 du code de `explorer`).

La deuxième phase consiste à envoyer les éventuels messages contenus dans la file d'attente à leur destinataire. Nous avons ici – même si ça n'est pas explicitement montré sur l'algorithme – utilisé une file d'attente par destinataire. Cette solution présente l'avantage de limiter les risques d'attente par rapport à l'utilisation d'une file unique.

Il est enfin nécessaire d'apporter une dernière précision concernant le communicateur qui n'apparaît pas sur l'algorithme présenté. Nous avons vu que le communicateur avait pour rôle de gérer les communications. Cependant les communications ne sont pas uniquement des opérations d'envoi et de réception de messages. A chaque fois, plusieurs opérations sont effectuées : pour un envoi d'état, il est nécessaire d'encoder l'état à envoyer et éventuellement de vérifier qu'il ne se trouve pas dans le cache d'émission, de même, à la réception d'un état, une première opération consiste à décoder l'état puis à tester si cet état est présent ou non dans l'espace d'état avant d'entamer une DFS à partir de cet état. Il est alors intéressant de voir quelles opérations sont effectuées par le communicateur et quelles opérations ne le sont pas.

Les opérations d'encodage ainsi que de décodage d'un état peuvent tout à fait être effectuées par le communicateur puisque nous avons vu que les méthodes de compressions utilisant d'éventuelles données partagées ne sont pas utilisées lors de l'encodage d'un état à envoyer. Ici, le communicateur n'a besoin d'accéder à aucune donnée partagée et peut donc effectuer ces opérations. Dans la pratique, notons toutefois que seule l'opération de décodage est effectuée sans aucun coût pour les exploreurs. Dans le cas d'un état à envoyer, la structure de donnée utilisée pour représenter l'état à encoder est réutilisée par l'exploreur une fois l'état envoyé. Pour des raisons d'efficacité, il n'y a en mémoire qu'une seule structure de donnée pour représenter les états dans la pile de recherche. Le franchissement de transitions ne crée jamais une nouvelle structure de donnée mais la modifie uniquement. Lors de l'ajout de l'état dans la file d'envoi, il est alors nécessaire que l'exploreur duplique cette structure de donnée pour éviter toute lecture invalide par le communicateur.

Considérons maintenant l'opération d'accès au cache. Ici, il n'est absolument pas nécessaire de partager le cache d'émission. Le communicateur peut donc y accéder sans aucun risque. Ce n'est pas le cas de la vérification effectuée à la réception d'un état. Dans ce cas, il est nécessaire d'accéder à l'espace d'état. L'intérêt de notre approche étant de bien dissocier les rôles des deux types de processus, nous déléguons donc cette opération aux exploreurs.

5.4 Seconde approche : communications partagées

Nous proposons ici de changer l'approche pour utiliser tous les *threads* de la même façon pour leur permettre à la fois de communiquer et d'explorer l'espace d'état.

5.4.1 Gestion des communications

Comme nous pouvons le voir sur la figure 5.5, cette nouvelle architecture nous oblige à protéger les *buffers* de communications en plus de l'espace d'état.

Remarque L'implémentation du standard MPI utilisée par Cyclades n'est malheureusement par parfaitement adaptée aux systèmes multithreadés. Ainsi, il est nécessaire de garantir l'accès exclusif aux primitives de communications en émission mais également en réception. Plus simplement, il n'est pas possible à un *thread* d'émettre un message alors qu'un autre *thread* effectue simultanément une réception de message. Cette limitation nous oblige à adapter nos algorithmes. Nous présenterons toutefois les modifications à apporter pour permettre l'adaptation des algorithmes à une implémentation de MPI autorisant la concurrence d'accès aux buffers d'émission et de réception.

Cet accès exclusif aux *buffers* de communications nous oblige à être très prudent quant aux opérations de communications et également au comportement des processus. Ainsi, les *threads* ne pourront plus se mettre en attente de messages entrants sur des appels bloquants ; ceci bloquerait l'accès aux primitives de communication pour tous les autres processus.

Pour résoudre ces problèmes de communications, il nous faut alors utiliser de préférence les appels aux primitives non bloquantes pour permettre aux *threads* de relâcher les verrous au plus tôt. Pour mettre les *threads* en attente, nous avons utilisé les signaux comme mécanisme de synchronisation. Les accès aux primitives étant tout de même assez fréquents il nous a paru inefficace de recourir à des mécanismes d'attente active.

5.4.2 Détection de la terminaison

Ici, la détection de la terminaison est assez simple à gérer. Un nœud est considéré comme inactif lorsque tous ses *threads* sont inactifs. Contrairement à l'architecture présentée précédemment, la seule source d'activité des messages provient du *buffer* de réception.

Nous utilisons alors le même mécanisme de variable partagée pour sauvegarder le nombre de processus inactifs. Ici, lorsqu'un processus détecte que tous les processus sont inactifs, il notifie le *manager* puis se met en attente de message entrant. C'est le seul cas dans lequel un processus peut se mettre en attente sur une primitive de communication.

5.4.3 Comportement des processus

Le comportement des processus dans cette architecture où les communications sont partagées est présenté à la figure 5.6.
La procédure `main` est la procédure principale. C'est le seul endroit où une différence est effectuée entre les différents *threads*. Au début de l'exécution, tous les *threads* sont considérés comme inactifs. Le *thread* principal ($id = 0$) peut alors se mettre en attente de message entrant *via* un appel à une primitive de communication bloquante (fonction `Receive` dans la procédure `listen`) tandis que les autres *threads* se placent tous en attente d'un signal (`start`). Cette attente garantit au *thread* principal qu'il sera le seul *thread* à recevoir le premier message entrant.

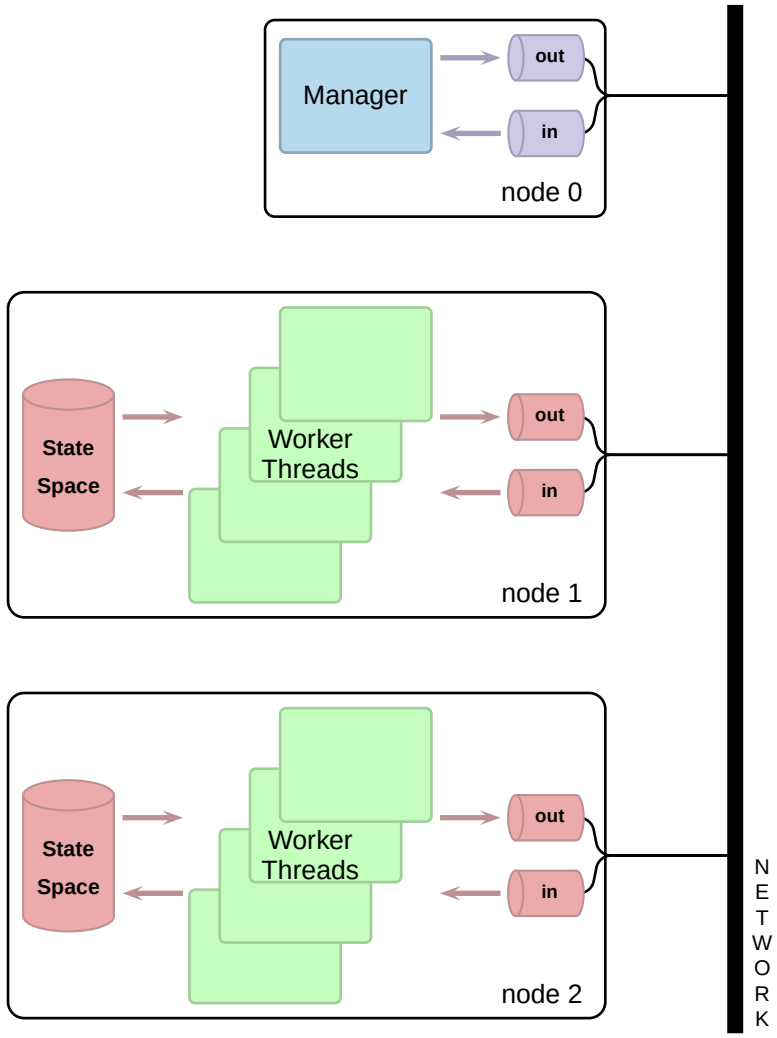


FIG. 5.5 – Architecture répartie multithreadée sans communicateur dédié (4 threads par nœud)

```
main ()
1   mutex_lock(MPI_mutex);
2   if id = 0 then
3      listen();
4      cond_signal(start);
5   else
6      cond_wait(start);
7      test_incoming();
8   end if
9   mutex_unlock(MPI_mutex);
10
11  exploration();

listen ()
1   // Blocking call
2   global_idle++;
3   msg ← Receive();
4   global_idle−−;

test_incoming ()
1   // Non blocking call
2   IProbe();
3   if no incoming message then
4      global_idle++;
5      cond_wait(MPI_mutex);
6      global_idle−−;
7   end if
8   msg ← Receive();

wake_up ()
1   mutex_lock(MPI_mutex);
2   IProbe();
3   if incoming message then
4      cond_signal(MPI_mutex);
5   end if
6   mutex_unlock(MPI_mutex);

exploration ()
1   while not termination do
2
3      while msg ≠ ∅ do
4         s ← Get_State(msg);
5         DFS(s);
6      end while
7
8      mutex_lock(MPI_mutex);
9      if global_idle=NB_THREADS−1 and no incoming then
10        send(idle);
11        listen();
12     else
13        test_incoming();
14     end if
15     mutex_unlock(MPI_mutex);
16  end while
```

FIG. 5.6 – Comportement des processus sur chaque nœud.

Ce mécanisme permet uniquement de sélectionner un unique *thread* pour effectuer l'appel à la primitive de réception bloquante pour éviter tout risque d'interblocage.

La procédure d'exploration est assez simple. Elle consiste dans un premier temps à extraire tous les états à explorer contenus dans le message entrant et à démarrer une exploration à partir de chacun de ces états. Une fois ces explorations terminées, il faut ensuite détecter s'il on est le dernier *thread* actif (test de la ligne 9, procédure `exploration`). Dans ce cas, on notifie alors le *manager* puis on peut se mettre en attente de message entrant *via* un appel à la primitive de réception bloquante.

Dans le cas où le *thread* n'est pas le dernier *thread* actif, la procédure `test_incoming` est appelée. Dans cette procédure, le premier appel est un appel non bloquant permettant de savoir si un message entrant est disponible. Dans ce cas, on peut alors directement le recevoir (ligne 8). Dans le cas contraire, le *thread* est alors inactif ; il modifie alors la valeur de la variable globale puis se met en attente d'un signal qui libère alors le verrou exclusif d'accès aux primitives de communications.

Notons que les *threads* en attente de signal ne sont pas directement réveillés par des messages entrants mais par le (ou les) *thread* actif. Ils sont alors réveillés lorsque le *thread* actif a détecté qu'un message entrant est disponible. Régulièrement, les *threads* actifs accèdent alors aux primitives de communications dans l'unique but de détecter si un message est arrivé. Dans ce cas, il peuvent alors libérer un *thread* qui serait éventuellement en attente. C'est le rôle de la fonction `wake_up`.

5.5 Gestion de l'espace d'état

5.5.1 Accès à l'espace d'état

Dans cette version de Cyclades, l'accès à l'espace d'état s'effectue de manière absolument identique à la version uniquement parallèle. Les mêmes politiques de verrouillages sont utilisées.

5.5.2 Représentation et compression d'état

Là encore, aucune modification n'est nécessaire pour gérer les différentes techniques de représentation et de compression d'état. Il n'y a ici aucune interférence entre les mécanismes d'adaptation liés à la répartition ou à la parallélisation.

Remarque sur les Δ-markings Une dernière question peut toutefois se poser quant à l'efficacité des Δ-*markings* dans ce nouvel environnement. Ici, il est cependant assez clair que les performances ne seront absolument pas dégradées. Le nombre de Δ-*markings* peut toutefois varier comparé à une solution uniquement répartie mais ceci est uniquement dû à l'impact du parcours sur la méthode des Δ-*markings*. Il n'y a cependant aucune dégradation véritable puisque toutes les explorations commencées à partir d'états reçus sont indépendantes les unes des autres. Le fait qu'elles soient traitées par le même processus ou par deux processus distincts n'a donc aucun impact sur le résultat final.

5.6 Expérimentations préliminaires

Des expérimentations préliminaires ont été effectuées pour les deux architectures présentées. Les résultats sont présentés sur le tableau 5.1. Ces résultats ont été obtenus en utilisant deux machines multicore : une quadri-core et une octo-core. Nous avons donc été en mesure de simuler 3 sites distants sur lesquels tournaient jusqu'à 4 *threads* ou encore 4 sites distants sur lesquels tournaient 3 *threads*. L'implémentation de cette version parallèle et répartie de Cyclades étant pour l'instant encore en phase de développement, ces premiers résultats ont été obtenus à l'aide d'une première version de test. Les résultats doivent donc être analysé avec le recul et la prudence nécessaire.

La figure 5.1 présente ces premiers résultats obtenus grâce à cette configuration expérimentale. Ils ont été obtenus en utilisant un partitionnement basé sur la détection de cycle et la méthode des Δ-*markings* avec une taille maximale des séquence reconstituantes fixée à 10. La colonne n représente le nombre de nœuds simulés et t le nombre de *threads* utilisés sur chacun de ces nœuds. La colonne *Single communicator* représente les résultats obtenus en utilisant la première solution architecturale à savoir l'extension d'Eddy_Murφ. La colonne *Shared communications* recense les résultats obtenus par l'utilisation de la seconde solution architecturale. Pour chaque test nous avons reporté le temps d'exécution (en secondes) ainsi que le gain obtenu par rapport à la version répartie simple entre crochet.

n	t	*Single communicator*	*Shared communications*	*Single communicator*	*Shared communications*
		Dining philosophers 43.10^6 *states* – 611.10^6 *arcs*		*Leader election protocol* 27.10^6 *states* – 328.10^6 *arcs*	
2	1	16 729 [1.00]	16 729 [1.00]	6 267 [1.00]	6 267 [1.00]
	2	15 231 [1.10]	11 781 [1.42]	5 917 [1.06]	4 263 [1.47]
	3	9 726 [1.72]	7 435 [2.25]	3 560 [1.76]	2 956 [2.12]
	4	6 534 [2.56]	5 748 [2.91]	2 401 [2.61]	2 262 [2.77]
3	1	10 646 [1.00]	10 646 [1.00]	3 701 [1.00]	3 701 [1.00]
	2	9 789 [1.09]	8 004 [1.33]	3 525 [1.05]	2 643 [1.40]
	3	6 781 [1.57]	5 021 [2.12]	2 229 [1.66]	1 832 [2.02]
	4	4 549 [2.34]	3 857 [2.76]	1 463 [1.53]	1 445 [2.56]
4	1	8 150 [1.00]	8 150 [1.00]	2 338 [1.00]	2 338 [1.00]
	2	7 742 [1.05]	6 520 [1.25]	2 185 [1.07]	1 758 [1.33]
	3	6 269 [1.30]	4 553 [1.89]	1 489 [1.57]	1 328 [1.76]
	4	–	–	–	–
		Eisenberg & McGuire 33.10^6 *states* – 165.10^6 *arcs*		*Sieve of Eratosthene* 30.10^6 *states* – 173.10^6 *arcs*	
2	1	1 252 [1.00]	1 252 [1.00]	1 506 [1.00]	1 506 [1.00]
	2	1 123 [1.11]	955 [1.31]	1 415 [1.06]	991 [1.52]
	3	772 [1.62]	591 [2.12]	880 [1.71]	717 [2.10]
	4	506 [2.47]	467 [2.68]	560 [2.69]	548 [2.75]
3	1	736 [1.00]	736 [1.00]	1 074 [1.00]	1 074 [1.00]
	2	692 [1.06]	608 [1.21]	982 [1.09]	808 [1.33]
	3	493 [1.49]	413 [1.78]	651 [1.65]	540 [1.99]
	4	346 [2.13]	299 [2.46]	421 [2.55]	418 [2.57]
4	1	610 [1.00]	610 [1.00]	787 [1.00]	787 [1.00]
	2	574 [1.06]	521 [1.17]	729 [1.07]	702 [1.12]
	3	504 [1.21]	394 [1.55]	489 [1.61]	435 [1.81]
	4	–	–	–	–

Tab. 5.1 – Expérimentations préliminaires

Une première observation globale concernant cette seconde solution architecturale nous permet de voir qu'elle n'est à priori pas forcément très efficace. Les gains de temps obtenus sont assez limités. Nous pensons que ceci est principalement du à la gestion des communications assez limitées dans un contexte parallèle. La politique de verrouillage semble ici extrêmement lourde et coûteuse. Pour

pouvoir plus aisément conclure sur cette méthode, il serait nécessaire de développer une librairie de communication plus souple pour le multithreading.

Concernant l'extension d'Eddy_Murφ, les premiers résultats sont encourageants. Dans la plupart des cas, des gains significatifs sont obtenus. La politique de verrouillage étant beaucoup plus optimisée dans cette solution, le parallélisme est clairement meilleur même s'il n'est pas optimal. Notons qu'ici, sur les t *threads* utilisés, un *thread* est dédiés uniquement à la gestion des communications. Cette gestion étant assez limitée il est plus intéressant de comparer les temps en considérons non pas t *threads* mais à chaque fois $t-1$.

A priori, l'utilisation d'une unique file partagée ne semble pas dégrader significativement les performances et paraît être compensée par le coût de décodage des états reçus ainsi que les reconstitution liées à l'utilisation de la méthode des Δ-*markings*.

Tous ces résultats nécessiteraient d'être effectués plus en profondeur. Il serait également nécessaire d'essayer d'utiliser – ou de développer – une bibliothèque de communication plus adaptée au multithreading. Enfin, le code demanderait à être également amélioré.

5.7 Conclusion

Cette architecture à la fois répartie et parallèle nous semble particulièrement intéressante pour le model checking. C'est cette architecture qui semble proposer le plus de possibilités et de souplesse. Les premières expérimentations effectuées tendent à nous faire penser que le gain de temps peut être réel en parallélisant la version répartie de Cyclades. Cette version préliminaire semble pouvoir combiner les avantages des deux architectures.

La question de la file unique partagée se pose malgré tout dès lors que le nombre de *threads* sera plus important. Les expérimentations ne permettent pas pour l'instant de conclure sur ce point mais l'on peut penser que cette file sera la source d'une possible dégradation des gains. L'extension d'Eddy_Murφ pourrait alors se révéler la plus intéressante en déléguant au *thread* communicateur le soin d'effectuer un équilibrage de charge en dispatchant les messages à des files spécifiques à chaque *threads*.
Ce n'est cependant pas la seule raison pour laquelle nous trouvons cette solution particulièrement intéressante. Cette architecture parallèle et répartie nous permet d'envisager des évolutions futures différentes.

Nous avons vu que l'efficacité du model checking réparti est fortement dépendante du partitionnement sous-jacent de l'espace d'état. Tant en terme d'équité de la répartition qu'en terme de degré de localité. L'équilibrage de charge est une solution possible pour répondre à la question de l'équité. Cette méthode n'est cependant pas gratuite et nécessite généralement soit (1) de transmettre un grand nombre d'états sur le réseau lors d'un rééquilibrage – quitte à saturer le réseau – si l'on ne veut pas perdre une partie du travail déjà effectué, soit (2) de supprimer un grand nombre d'états dans les espaces d'états locaux ce qui peut aboutir à une exploration redondante de ce groupe d'états. Dans tous les cas, le coût du rééquilibrage n'est pas nul et peut être effectué plusieurs fois pour la vérification d'un même modèle.

En utilisant une architecture parallèle et répartie, on peut alors envisager d'exécuter, sur un même cluster, plusieurs approches de partitionnement réparti simultanément. En utilisant, par exemple, des machines *quadri*-processeurs, on peut lancer quatre approches de partitionnement sur le même réseau ; les quatre *threads* tournant sur chacune des machines parcourant chacun l'espace d'état en utilisant un partitionnement spécifique. Au fur et à mesure de l'exécution, il serait alors possible de stopper certaines approches de partitionnement jugées trop inefficaces (en se basant tant sur le degré de localité obtenu que sur l'équilibrage de la charge) pour multithreader les approches qui

semblent donner les meilleurs résultats. Cette approche pourrait donc se substituer plus ou moins aux méthodes d'équilibrage de charge classiques pour aboutir à de meilleures performances.

Chapitre

6

Réductions d'ordre partiel

Sommaire

L'exécution concurrente de plusieurs processus est une source majeure du phénomène d'explosion combinatoire qui survient lors d'une procédure de vérification exhaustive. Considérons un ensemble de p processus pouvant chacun être dans q états différents. Le système résultant de l'exécution concurrente de ces processus aura alors, au pire, un espace d'état de q^p états, ce qui le rend difficilement analysable tel quel même pour des petites valeurs de p et q. Ce phénomène d'explosion d'état est un frein considérable à l'utilisation pratique des méthodes de model checking sur des systèmes industriels.

Le problème de l'explosion combinatoire a été à la source de nombreuses recherches durant les deux dernières décennies et plusieurs techniques ont été proposées afin de lutter contre ce phénomène. Les techniques d'ordre partiel auxquelles nous nous intéressons dans ce chapitre en font notamment partie.. Celles-ci émanent de l'observation suivante : en raison de la sémantique d'entrelacement des systèmes concurrents, plusieurs exécutions différentes peuvent avoir le même effet sur le système

et n'être qu'une permutation d'une même séquence. Ainsi, un moyen efficace de réduire l'explosion d'état serait de n'explorer qu'une seule ou quelques unes de ces séquences et d'ignorer toutes les autres séquences qui leur sont équivalentes. C'est pourquoi le terme de model checking d'ensembles représentatifs (*model checking using representatives*, [50]) paraît plus adapté que celui d'"ordre partiel".

La technique des ensembles têtus (*stubborn sets*) d'Antti Valmari [54, 55, 53] est un membre de cette famille. La plupart de ces techniques sont basées sur un algorithme de recherche sélectif : à chaque état traité par l'algorithme, les transitions exécutées pour générer les successeurs de l'état sont sélectionnées selon certains critères qui garantissent que la propriété à vérifier sera préservée dans le graphe d'accessibilité réduit. L'ensemble des transitions ainsi calculé est dit têtu car les transitions qui ne sont pas dans cet ensemble ne peuvent pas affecter son comportement. En éliminant certaines transitions de cet ensemble têtu, certains états ne seront jamais explorés, et la taille de l'espace d'état peut diminuer de manière drastique. Dans le meilleur des cas, la réduction est exponentielle.

Il a été prouvé que le fait de déterminer si un ensemble est têtu en un état est au moins aussi difficile que le problème de l'accessibilité [30]. Les algorithmes de calcul d'ensembles têtus utilisent donc des conditions suffisantes qui assurent que l'ensemble calculé est bien têtu. Ces conditions dépendent du formalisme de haut niveau décrivant le système, p.ex., Promela, les réseaux de Petri, mais, indépendamment de ce formalisme, elles reposent sur la notion de dépendance entre transitions. Typiquement, lorsqu'une transition est ajoutée à l'ensemble têtu, l'algorithme doit déterminer les transitions qui peuvent potentiellement bloquer ou sensibiliser cette transition. Pour les réseaux de Petri ordinaires, ces dépendances peuvent être aisément détectées à partir de la structure du réseau. Pour les réseaux de Petri colorés, une méthode de détection des dépendances entre les transitions a été proposée dans [31].

Ces techniques d'ordres partiels sont parmi les méthodes les plus efficaces pour combattre l'explosion combinatoire. Comme nous venons de le préciser, ces méthodes se basent sur le modèles pour être calculées à la volée. Nous verrons cependant qu'une autre condition aux calculs des ordres partiels se base sur le parcours lui-même et notamment sur la pile de recherche. Cette méthode, efficace pour le model checking séquentiel, devient extrêmement ardue à maintenir dans un contexte de répartition où la cohérence du parcours n'existe plus que localement.

Nous avons travaillé, notamment avec Sami Evangelista, sur les méthodes d'ordre partiel. Ces travaux ont débouché sur plusieurs algorithmes qui s'avèrent être plus efficaces que les approches classiques dans les contextes répartis et dans un grand nombre de cas – voire dans tous les cas pour les propriétés de sûreté – dans un contexte séquentiel [41].

Ce chapitre se présente comme suit : après une présentation succincte des méthodes d'ordre partiel classiques, nous présenterons ces nouveaux algorithmes dans un contexte séquentiel. Dans une seconde partie, nous présenterons l'état de l'art des réductions d'ordre partiel en environnement réparti avant de décrire l'adaptation de notre algorithme séquentiel au contexte réparti. Nous verrons alors que l'algorithme d'ordre partiel pour les propriétés de sûreté s'adapte paradoxalement beaucoup moins bien que celui pour les propriétés de vivacité – pourtant plus renforcé – à la répartition. Ceci n'est finalement pas si paradoxal qu'au premier abord puisque l'algorithme présenté pour les propriétés de sûreté ne se base *que* sur la pile contrairement au second.

6.1 Les méthodes d'ordre partiel

Nous allons très rapidement rappeler les principales techniques d'ordre partiel, à savoir :
– la méthode des ensembles têtus de Valmari [55, 54, 53, 57, 58, 59],
– la méthode des ensembles persistants de Godefroid [45, 46, 44],

– la méthode des ensembles dormants de Godefroid [45, 46, 44],
– la méthode des pas couvrants de Vernadat, Azéma, et Michel [63, 64].
Par souci de simplicité, nous décrirons ces méthodes dans le contexte des réseaux de Petri ordinaires. Les différentes réductions fournies seront illustrées à l'aide du réseau de Petri de la figure 6.1.

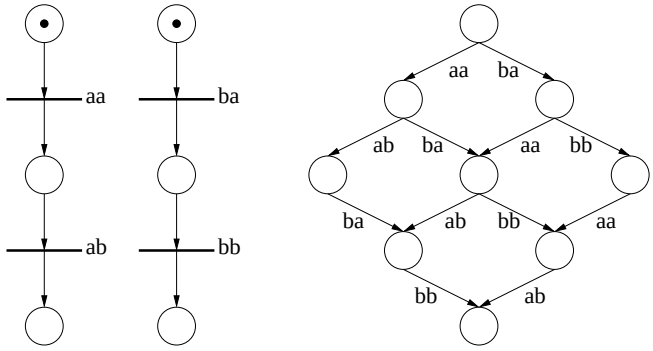


FIG. 6.1 – Un réseau de Petri et son graphe d'accessibilité

Toutes ces méthodes se basent sur la notion d'indépendance de transitions dont nous donnons ici la définition :

Définition 6.1 (Indépendance)**.** *Deux transitions t_1 et t_2 sont indépendantes en un état s si les deux conditions suivantes sont réunies :*

1. *Le franchissement de t_1 (respectivement t_2) en s ne peut ni bloquer ni permettre le franchissement de t_2 (respectivement t_1).*
2. *Si les deux transitions sont franchissables en s alors elles commutent : $s[t_1.t_2\rangle s'$ et $s[t_2.t_1\rangle s'$*

Si deux transitions sont indépendantes en tout état du système, alors on dit qu'elles sont *globalement indépendantes*. L'intuition derrière cette notion d'indépendance est que l'ordre d'exécution des transitions t_1 et t_2 en s n'est pas important : elles ne peuvent ni se bloquer ni se sensibiliser et l'ordre de franchissement de ces transitions n'a aucune incidence sur l'état global du système.

La méthode des ensembles persistent étant fortement similaire à celle des ensemble têtus, nous ne présenterons que cette dernière méthode. Notons également que c'est la méthode des ensemble têtu qui est implémentée dans Helena et que nous étendrons au contexte réparti dans les sections suivantes.

i. Les ensembles têtus

La technique des ensembles têtus de Valmari s'appuie sur un algorithme de recherche sélectif : à chaque état visité, seul un sous-ensemble des transitions franchissables sera considéré pour l'exploration des successeurs.

Les critères de sélection assurent que toute transition de l'ensemble calculé est têtu : les transitions qui ne sont pas dans cet ensemble ne peuvent pas affecter le comportement des transitions têtues. Cet ensemble est, en quelque sorte, clos par rapport aux interactions. Les transitions franchissables de cet ensemble le resteront donc après le franchissement d'une quelconque séquence de transitions non têtues. Le franchissement de ces transitions peut donc être avancé. Les transitions sensibilisées qui ne sont pas têtues sont alors ignorées par l'algorithme et leur franchissement est retardé.
Cette méthode tire parti du fait que dans un système concurrent, l'effet global d'un ensemble d'événements est souvent indépendant de l'ordre de production de ces événements. Nous appellerons *graphe réduit* le graphe obtenu par ce procédé.

Le choix de cet ensemble têtu dépend de la propriété à vérifier. Une variété d'algorithmes permettant de calculer des ensembles têtus pour diverses propriétés a été rappelée dans [144]. Par souci de simplicité nous nous intéressons à la définition suivante qui constitue une base commune à la majorité des algorithmes de calcul d'ensembles têtus.

Définition 6.2. *Soient N un réseau de Petri, $m \in \mathcal{M}$, et $S \subseteq T$. L'ensemble S est* **têtu** *en m si les deux conditions suivantes sont remplies :*

D1 $\forall \sigma \in T \setminus S^*, m[\sigma.t\rangle \Rightarrow m[t.\sigma\rangle$

D2 $\exists t \in T \mid m[t\rangle \Rightarrow \exists t \in S \mid \forall \sigma \in T \setminus S^*, m[\sigma\rangle \Rightarrow m[\sigma.t\rangle$.
La transition t est appelée transition clé de l'ensemble S.

La condition **D1** assure qu'une transition bloquée ne peut pas être sensibilisée par le tir d'une transition n'appartenant à l'ensemble têtu. Par la condition **D2**, il existe une transition clé t dans S qui est franchissable en m (en posant $\sigma = \epsilon$) et qui le reste tant qu'une transition de S n'est pas franchie.
Un ensemble têtu respectant les conditions **D1** et **D2** de la définition précédente est souvent qualifié de *dynamically stubborn* dans la littérature [60, 62, 144]. Le *dynamically* tient du fait que cette définition se base sur la sémantique du réseau, i.e., son graphe d'accessibilité. Elle est donc difficilement exploitable telle quelle par un algorithme.

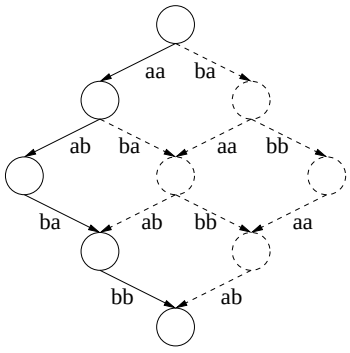


FIG. 6.2 – Le graphe d'accessibilité de la figure 6.1 réduit par la technique des ensembles têtus

La figure 6.2 décrit le graphe d'accessibilité du réseau de la figure 6.1 réduit par la technique des ensembles têtus. Les nœuds et arcs en pointillé ne sont pas visités par l'algorithme. Il apparaît que l'algorithme de recherche sélectif ne sélectionne qu'une unique transition pour chaque nœud. En examinant les transitions du réseau, ce fait n'est guère étonnant. En effet chacune de ces transitions possède une unique place en entrée qu'elle ne partage avec aucune autre transition. Nous voyons donc de manière évidente, que toute transition franchissable est à la fois une transition clé, et un ensemble têtu à elle seule.

ii. Les ensembles dormants

Nous nous intéressons maintenant à la méthode des ensembles dormants (*sleep sets*) développée par Godefroid dans [45, 46] puis dans sa thèse de doctorat [44]. Cette méthode s'appuie elle aussi sur la notion d'indépendance entre transitions mais procède différemment dans son mode de sélection des transitions. En effet, la technique des ensembles têtus / persistants tentent de "prédire" les conflits qui pourraient survenir entre les transitions du modèle alors que la technique des ensemble dormants utilisent des informations sur le passé de la recherche.
L'idée sous-jacente à cette technique est simple et peut être décrite à l'aide de la figure 6.3.
Soient m un marquage et t_1 et t_2 deux transitions franchissables et indépendantes en m. Par la définition de l'indépendance, nous avons $m[t_1.t_2\rangle m'$ et $m[t_2.t_1\rangle m'$. Soient m_1 et m_2 les marquages

définis par : $m[t_1\rangle m_1$ et $m[t_2\rangle m_2$. Supposons que l'algorithme de recherche procède en profondeur d'abord et exécute successivement t_1 puis t_2, remonte au marquage m_1 puis m et enfin exécute t_2 pour atteindre m_2. Le fait que t_1 et t_2 soient indépendantes en m nous garantit que le tir de t_1 est inutile puisque $t_1.t_2$ a déjà été franchie en m et que cette séquence est équivalente à la séquence $t_2.t_1$. Ce tir mènerai donc à un marquage déjà visité.
La méthode des ensembles dormants procède donc de la manière suivante. A chaque état e du système est associé un ensemble dormant qui est constitué de transitions franchissables en e, mais qui ne seront pas exécutées par l'algorithme. La relation d'indépendance garantit en effet que le tir de ces transitions mènerai à des états qui ont déjà été visités par l'algorithme. Nous laisserons au lecteur le soin de consulter l'algorithme complet de calcul des ensembles dormants dans [46, 44].

La figure 6.4 présente le graphe d'accessibilité du réseau de la figure 6.1 réduit par la méthode des ensembles dormants. Notons que le nombre de états n'a pas diminué. En revanche, de nombreuses transitions ne sont pas exécutées.

L'intérêt de la méthode des ensembles dormants n'est pas flagrant à la vue du graphe d'accessibilité réduit de la figure 6.4. Nous constatons en effet que le nombre d'états visités par l'algorithme n'a pas diminué. Cette méthode est pourtant très utile, et ce pour trois raisons.
La technique des ensembles dormants permet tout d'abord de réduire considérablement le nombre de tests d'appartenance à l'espace d'état. Ce test intervient lorsqu'un état est visité durant la recherche afin de déterminer si il doit être exploré ou non. Le nombre de tests effectués est égal au nombre d'arcs du graphe d'accessibilité : à chaque fois qu'une transition est exécutée, un état est atteint et l'algorithme doit déterminer si cet état a déjà été visité. Ce test est très coûteux puisqu'il met en oeuvre diverses opérations : codage de l'état, insertion dans une table de hachage, comparaison de vecteurs de bits en cas de conflit dans la table... En réduisant le nombre d'arcs cette méthode peut donc accélérer la recherche de manière significative.
Un algorithme de recherche sélectif utilisant la technique des ensembles têtus / persistants peut parfois sélectionner des transitions indépendantes en un état puisque la définition de ces ensembles est basée sur la notion d'indépendance globale (alors que la technique des ensembles dormants repose sur la définition d'indépendance locale). La technique des ensembles dormants peut alors être utile pour éviter l'exploration de multiples entrelacements de ces transitions indépendantes et réduire plus encore le nombre d'états visités. La combinaison de ces deux techniques a été étudiée dans [46, 44, 61].
Enfin la technique des ensembles dormants est particulièrement utile lorsqu'elle est combinée avec la technique de cache d'état (*state space caching*). Rappelons que cette dernière est une méthode de représentation de l'espace d'état. L'idée sous-jacente est de ne garder en mémoire que les états de la pile afin de prévenir l'entrée dans un cycle. L'augmentation du temps d'exécution peut être brutale puisqu'un état sera visité pour chaque chemin menant de l'état initial à l'état en question. En réduisant fortement l'entrelacement, la technique des ensembles dormants peut limiter les explorations répétées d'un même état. Considérons par exemple le graphe d'accessibilité de la figure 6.4. Nous constatons que chaque état du système est relié à l'état initial par un seul chemin. Un algorithme de recherche combinant ces deux méthodes ne visitera pas donc pas plus d'une fois

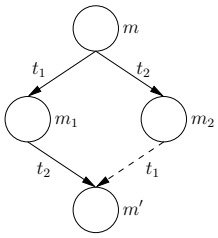


FIG. 6.3 – Le tir de la transition t_1 est inutile en m_2

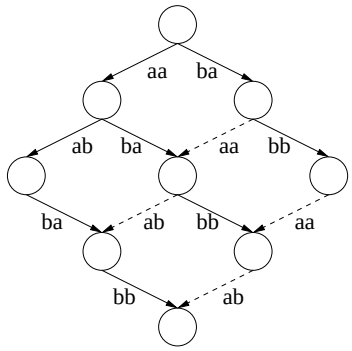


FIG. 6.4 – Le graphe d'accessibilité de la figure 6.1 réduit par la technique des ensembles dormants

le même état. L'utilisation conjointe de ces deux techniques a été étudiée dans [34].

iii. Les pas couvrants

Contrairement aux méthodes de Valmari et Godefroid, la méthodes des pas couvrants [63, 64] ne repose pas sur un algorithme de recherche sélectif. A chaque état rencontré, toutes les transitions sensibilisées seront effectivement franchies. La différence repose dans le fait que l'algorithme répartit les transitions sensibilisées en deux sous-ensembles : les transitions à explorer de façon standard, i.e., comme dans un algorithme de recherche classique, et des transitions indépendantes qui sont groupées et franchies atomiquement. L'algorithme exécute donc des pas : des séquences de transitions indépendantes.

La figure 6.5 présente le graphe d'accessibilité du réseau de la figure 6.1 réduit par la méthode des pas couvrants. Dans l'état initial les transitions aa et ba sont toutes deux indépendantes. L'algorithme franchit donc le pas aa.ba. L'ordre de franchissement de ces transitions n'a évidemment aucune importance puisqu'elles sont indépendantes. Dans l'état atteint les transitions ab et ba sont elles aussi indépendantes et franchies atomiquement.
Il est à noter que la méthodes des pas couvrants ne construit pas un sous-graphe du graphe initial : les arcs de ce graphes étiquetés par des pas ne figurent pas dans le graphe initial.

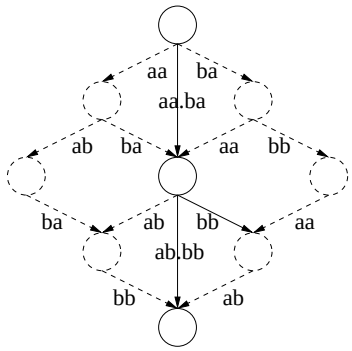


FIG. 6.5 – Le graphe d'accessibilité de la figure 6.1 réduit par la technique des pas couvrants

Enfin, notons qu'il est possible de combiner la méthode des pas couvrants et la méthode des pas persistants [51, 52]. Cette combinaison est théoriquement meilleure que chacune de ces méthodes car pour tout graphe persistent (le graphe réduit par la technique des ensembles persistants) et pour tout graphe de pas couvrant, il existe un graphe de pas persistants (le graphe réduit par l'utilisation conjointe des deux méthodes) plus petit.

6.1.1 Conservation des propriétés

Un algorithme de recherche sélectif utilisant les méthodes décrites précédemment peut être utilisé pour construire un graphe d'accessibilité réduit. Ce graphe inclut tous les marquages morts du graphe initial ainsi qu'une séquence de transitions infinie si une telle séquence existe [60, 62]. Malheureusement en raison du phénomène d'ignorance, la méthode des ensembles têtus ne peut guère faire mieux.
Considérons le réseau de la figure 6.6. L'ensemble constitué de l'unique transition t est têtu pour le marquage initial. En effet cette transition est franchissable et ne partage pas ses places en entrée. C'est donc une transition clé et un ensemble têtu à elle seule. Le franchissement de cette transition ne change pas le marquage. Le graphe réduit est donc réduit à un unique nœud. Or il apparaît clairement que le graphe réduit ne peut pas être utilisé pour valider toutes sortes de propriétés. En particulier, une propriété portant sur les places du sous-réseau R ne pourra pas être validée à l'aide du graphe réduit. Ce problème est généralement appelé "phénomène d'ignorance" (*ignoring phenomenon* [57]). Le comportement d'un processus peut être totalement ignoré à partir de certains états atteint par l'algorithme de recherche sélectif.

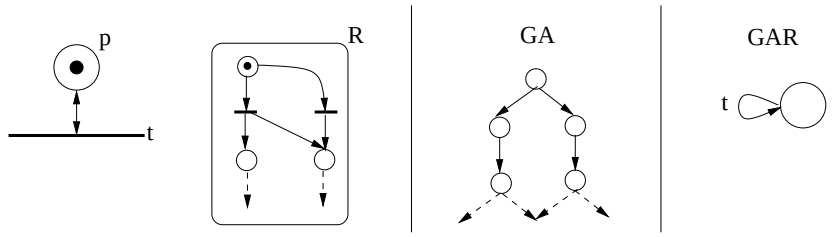


FIG. 6.6 – Illustration du phénomène d'ignorance. Les transitions du sous-réseau R sont systématiquement ignorées lors de la construction du graphe réduit GAR.

Les réductions d'ordre partiel [56, 50, 44] réduisent l'exploration de l'espace d'état en réduisant le nombre de chemins à considérer pendant la vérification. Le principe sous-jacent consiste à sélectionner, pour chaque état visité, un sous-ensemble d'actions exécutables qui seront considérées tans que d'autres seront retardées ou déléguées à un état futur. Pour conserver différentes propriétés, il est nécessaire que la sélection de ce sous-ensemble respecte un certain nombre de contraintes. Ces contraintes seront différentes selon le type de propriétés que l'on souhaite conserver.
Le mécanisme de sélection du sous-ensemble d'actions à considérer se base sur la notion de *fonction de réduction.*

Définition 6.3 (STG)**.** *Un graphe état-transition (state transition graph – STG) est un tuple $(S, s_0, A, \rightarrow)$ où S est un ensemble fini d'états ; $s_0 \in S$ est l'état initial du système ; A est un ensemble d'actions ; $\rightarrow \subseteq S \times A \times S$ est la relation de transition telle que $(s, a, s') \in \rightarrow \wedge (s, a, s'') \in \rightarrow \Rightarrow s' = s''$.*

Définition 6.4 (Fonction de réduction)**.** *Soit $(S, s_0, A, \rightarrow)$ un STG. Une fonction de réduction r est une projection de S vers 2^A telle que $\forall s \in S, r(s) \subseteq en(s)$.*

Lorsque $en(s) = r(s)$ pour un état s, la fonction n'aboutit à aucune réduction. On dit alors que l'état s est *totalement étendu.* Dans le cas contraire, l'état s est dit *partiellement étendu.* Une action a est considérée comme *ignorée* pour s si et seulement si $a \in en(s) \setminus r(s)$.
En utilisant une telle fonction de réduction lors de l'exploration de l'espace d'état, on construit alors un *graphe réduit.*

Définition 6.5 (STG Réduit)**.** *Soient $(S, s_0, A, \rightarrow)$ un STG et r une fonction de réduction. Le STG réduit $(S_r, s_{0_r}, A_r, \rightarrow_r)$ est défini par :*
- *$s_{0_r} = s_0$, $A_r = A$.*
- *$s \in S_r$ si et seulement s'il existe une séquence d'exécution finie $s_0 \stackrel{a_0}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} s_n$ telle que $s = s_n$ et $a_i \in r(s_i), \forall s_i \in \{s_0 \ldots s_{n-1}\}$.*

– $(s, a, s') \in \rightarrow_r$ *si et seulement si* $s \in S_r$, $(s, a, s') \in \rightarrow$ *et* $a \in r(s)$.

a. Réductions d'ordre partiel pour la détection d'états morts

Comme nous l'avons précisé précédemment, une fonction de réduction doit respecter un certain nombre de contraintes pour préserver certaines propriétés du graphe initial dans le graphe réduit. Ces contraintes varient selon le type de propriétés considérées. Cependant, le principe général de la théorie des réductions d'ordre partiel exploitant la commutativité d'actions concurrentes afin de limiter l'entrelacement superflu, toutes ces contraintes se basent sur la notion clé d'*indépendance* d'actions. Intuitivement, deux actions a et b sont indépendantes si elles ne peuvent se limiter entre elles et si elles peuvent commuté en tout état du système.

Définition 6.6 (Indépendence)**.** *Une relation d'indépendance est une relation symétrique et anti-reflexive* $I \in A \times A$ *satisfaisant les deux conditions suivantes pour tout état* $s \in S$ *et pour chaque actions* $(a, b) \in I$.

Franchissabilité *si* $a, b \in en(s)$ *et* $s \xrightarrow{b} s'$ *alors* $a \in en(s')$.

Commutativité *si* $a, b \in en(s)$ *alors* $s \xrightarrow{a} s'' \xrightarrow{b} s'$ *et* $s \xrightarrow{b} s''' \xrightarrow{a} s'$.

Deux actions a et b sont indépendante si et seulement si $(a, b) \in I$. Dans le cas contraire, elles sont considérées comme *dépendantes* et le couple (a, b) appartient à la relation $(A \times A) \setminus I$.
Cette relation d'indépendance est généralement vérifiée lors de la phase de compilation, i.e., avant l'exploration de l'espace d'état à l'aide d'une simple analyse statique du modèle. Typiquement, une action qui ne manipule que des variables globales (affectation, ...) est considérée comme indépendante de tout autre action.
Nous pouvons alors présenter les deux conditions qui permettent de construire un ensemble persistent (*persistent set* – PS) de transitions pour un état s.

C0 $r(s) = \emptyset$ si et seulement si $en(s) = \emptyset$.

C1 toute action dépendant d'une action de $r(s)$ ne peut être exécutée sans qu'une transition de $r(s)$ n'ait été exécutée avant.

Une fonction de réduction qui calcule de tels ensembles persistants préserve tous les états morts du systèmes [43] et peut donc être utilisée pour détecter de tels états. Informellement, la condition C0 garantit que l'algorithme de recherche utilisant la fonction de réduction continue à progresser si l'algorithme normal le fait.
La condition C1 permet de garantir qu'après l'exécution de n'importe quelle séquence contenant uniquement des transitions hors de $r(s)$, toutes les transitions de $r(s)$ seront toujours exécutables. On peut alors les exécuter immédiatement et retarder l'exécutions des autres transitions.

b. Réductions d'ordre partiel pour les propriétés de sûreté

Un algorithme de recherche utilisant les ensembles persistants peut éventuellement retarder le franchissement de certaines transition infiniment et ainsi ne pas explorer des états pertinents. Ce phénomène appelé *phénomène d'ignorance* (*action ignoring*)[56] peut être résolu par le biais d'une contrainte appelée *proviso*.

C2$^{\mathbf{S}}$ Soit un état $s \in S_r$, $a \in en(s)$ il existe un état s' accessible depuis s dans le graphe réduit, tel que $a \in r(s')$.

Cette condition assure que toute transition franchissable sera exécutée à partir d'un état accessible depuis s. Si la fonction de réduction satisfait cette condition simple, on peut démontrer que le graphe réduit est alors ce que Godefrois appelle un *trace automaton*. Les automates de trace (*trace automata*) ont comme propriété la conservation de l'accessibilité des états locaux : si un processus peut atteindre un état donné dans le graphe initial, alors il sera également en mesure de l'atteindre dans le graphe réduit. Les automates de trace peuvent alors être utilisés pour vérifier un grand nombre de propriété de sûreté comme, par exemple, des assertions sur des variables locales.

c. Réductions d'ordre partiel pour les propriétés de vivacité

Pour préserver les propriétés de vivacité, il faut garantir que chaque cycle du graphe d'accessibilité ne contient aucune transition franchissable qui ne sera jamais exécutée (pour chaque état du cycle). Cette contrainte entraîne le renforcement de la version du *proviso* noté $C2^L$.

C2^L Un cycle n'est pas permis s'il contient un état pour lequel une action a est exécutable mais jamais inclue dans $r(s)$ pour chaque état s du cycle.

Dans la pratique, cette condition est généralement remplacée par la condition suivante, induite par la condition C1, et beaucoup plus aisément implémentable :

C2$^{L'}$ Le long d'un cycle du graphe réduit, il y a au moins un état s totalement étendu.

Couplée à une autre condition (voir [39]) qui préserve l'entrelacement d'actions pertinentes (appelée actions *visibles*), le *proviso* $C2^L$ peut être utilisé pour construire des ensembles *ample* [50]. Une fonction de réduction utilisant de tels ensembles aboutit à la création d'un graphe réduit qui conserve les propriétés LTL-X du graphe initial.

6.1.2 Etat de l'art

Les *provisos* pour les propriétés de sûreté et de vivacité sont exprimées comme des propriétés du graphe réduit alors que l'on souhaite effectuer les réductions à la volée. Pour cela, on les reformule généralement en conditions qui peuvent être vérifiée pendant la construction du graphe réduit et qui sont, par conséquent, fortement liées à la façon dont on parcourt ce graphe ainsi qu'aux structures de données utilisées par le parcours.
Pour la recherche en profondeur (*Depth First Search* – DFS), on utilise généralement le fait que chaque cycle contient une transition atteignant la pile de recherche à un point donné de la recherche. Cette propriété est suffisante pour interdire qu'un état partiellement étendu ne franchisse une transition atteignant la pile de parcours. Cette contrainte donne une première version du *proviso*, notée $C2^L_s$ [47]. Ce *proviso* est notamment utilisé dans le model checker Spin [118].

C2^L_s Si $r(s) \neq en(s)$ alors aucune action de $r(s)$ ne peut atteindre un état de la pile.

Pour les propriétés de sûreté, une contrainte plus faible peut être utilisée. On peut notamment permettre à une transition d'atteindre un état dans la pile tant qu'une autre transition atteint un état en dehors de cette pile [43].
Pour la recherche en largeur (*Breadth First Search* – BFS), une version similaire utilisant la file de parcours a été récemment présentée dans [38].

C2^L_q Si $r(s) \neq en(s)$ alors aucune action de $r(s)$ ne peut atteindre un état de la file.

L'intuition derrière cette condition est qu'il n'est pas nécessaire de s'inquiéter de problème d'ignorance d'actions de s puisque l'on peut déléguer ce problème aux successeurs de s qui appartiennent tous à la file et qui seront explorés ultérieurement. Là encore, un contrainte réduite peut être utilisée pour aboutir à un *proviso* noté $C2^S_q$ pour les propriétés de sûreté qui requiert qu'au moins une action atteint un état dans la file.
Cette idée a été généralisé dans [37] pour les algorithmes d'exploration de l'espace d'état, c'est-à-dire, n'importe quel algorithme explicite qui partitionne l'espace d'état en trois ensembles disjoints : les *états ouverts* qui ont été explorés mais pas encore étendus, les *états fermés* qui ont été explorés et étendus (et qui peuvent être potentiellement ré-ouverts), et les *états non visités* qui n'ont pas encore été explorés. Ce nouveau *proviso* peut, par exemple, être utilisé dans le model checking dirigé [40].
Un état ouvert (ou non visité) est un état qualifié de *sûr* (*safe*) dans la mesure où il peut être atteint par un état partiellement étendu sans risque d'introduire un phénomène d'ignorance : la résolution de ce phénomène est délégué aux états qui seront explorés par la suite. Un état fermé, quant à lui, est un état dangereux puisqu'il a déjà été exploré.
Dans [48], une nouvelle approche est proposée qui consiste à préparer toutes les réductions de façon statique et donc, avant l'exploration. Cette méthode est alors indépendante de l'algorithme de recherche utilisé. Si l'on considère un système concurrent, qui est une composition de processus

séquentiels, les auteurs utilisent le fait qu'un cycle dans l'espace d'état est le résultat d'un cycle dans le modèle analysé (et plus précisément dans le comportement d'un processus séquentiel). L'idée est alors de déterminer de façon statique une action de ces cycles du modèle et de les identifier. Ces transitions particulières sont appelées *sticky transitions*. Le *proviso* peut alors être réduit à la condition suivante : un ensemble persistent qui n'inclut pas toutes les transitions franchissables ne peut contenir une *sticky transition*. En d'autres termes, le franchissement d'une *sticky transition* ne peut être effectué à partir d'un état partiellement étendu.
L'algorithme *two-phase* présenté dans [49] utilise une recherche différente dans la pile. Cet algorithme alterne deux phases, l'une dans laquelle il étend complètement les états exploré et l'autre dans laquelle des ensembles persistants réduits à des singletons sont calculés. Pour certains modèles, cet algorithme peut aboutir à des réductions plus efficaces que celle obtenues par le *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$. Nous reviendrons un peu plus en détail sur ce *proviso* dans 6.3.2.

6.2 Nouveaux provisos

6.2.1 Motivations

Comme nous l'avons expliqué précédemment, les méthodes d'ordre partiel peuvent réduire d'un facteur potentiellement exponentiel les états explorés lors de la vérification. Cependant, dans un certain nombre de cas, les réductions obtenues ne sont pas aussi importantes que celles que l'on pourrait attendre. Ceci est principalement du à deux facteurs.

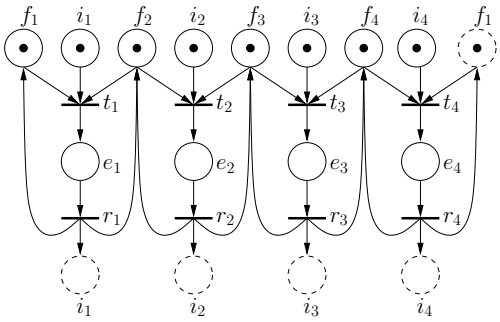
Premièrement, le calcul des ensembles persistants repose sur une analyse statique du modèle qui peut, dans certains cas, aboutir à des approximations trop importantes. Les réductions d'ordre partiel dynamiques ont été récemment proposées par Flanagan et Godefroid [42] pour essayer de limiter ce problème.

Une autre source d'inefficacité provient de la résolution du problème d'ignorance. Il existe en effet plusieurs modèles pour lesquels le proviso classique $\mathrm{C2}^{\mathrm{L'}}$ basé sur la recherche dans la pile produit de faibles résultats en terme de réductions. Nous illustrons ce problème à l'aide du réseau de Petri présenté sur la figure 6.7(a). Ce réseau modélise le problème du dîner des philosophe dans lequel un philosophe prend deux fourchette de façon atomique. Par souci de clarté, quelques places ont été dupliquées – elles sont représentées par des pointillés. Les places i_1, i_2, i_3 et i_4 modélisent l'état inactif des quatre philosophes tandis que les places e_1, e_2, e_3 et e_4 modélisent l'état dans lequel les philosophes sont en train de manger. Enfin, chaque place f_i modélise les fourchette du philosophe i. Pour s'asseoir à la table (transition t_i), le philosophe i doit prendre sa fourchette f_i et celle de son voisin, i.e., f_j où $j = i \mod n+1$. Une fois qu'il a fini de manger, il retourne dans l'état inactif et rend les deux fourchettes (transitions r_i).

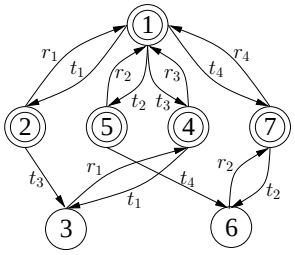
L'espace d'état généré pour ce modèle avec le *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ est reporté sur la figure 6.7(b). Les états totalement étendus sont cerclés deux fois. Les états sont également numérotés selon l'ordre de leur exploration par l'algorithme. Il apparaît que la réduction obtenue n'est pas optimale puisqu'elle ne fait l'économie que de l'exécution de deux transitions. En effet, le *proviso* aboutit généralement à la détection d'un cycle ce qui aboutit à une exploration complète d'une grande partie des états. Un *proviso* optimal (présenté figure 6.7(c)) n'introduirait aucun état inutile dans l'espace d'état réduit puisque l'ensemble persistent serait réduit à l'état initial totalement étendu.

Pour quatre philosophe, ce *proviso* optimal ne réduit l'espace d'état que de deux états. S'il on généralise le problème à n philosophes, la réduction est beaucoup plus importante. L'espace d'état complet et l'espace d'état réduit à l'aide du *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ auront tous les deux une taille en $\mathcal{O}(2^n)$ alors que le *proviso* optimal limitera l'espace d'état à $n+1$ états.

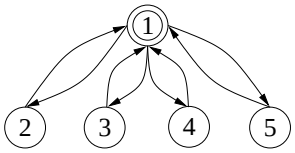
Notre intuition est que le problème d'ignorance apparaît rarement en pratique. En adoptant une



(a) Le réseau de Petri modélisant le dîner des philosophes



(b) PS + $C2^{L}_{s}$



(c) PS + proviso optimal

FIG. 6.7 – Un exemple illustrant nos motivations.

stratégie trop défensive, les implémentations traditionnelles du *proviso*– telles que celles basées sur une recherche dans la pile – introduisent beaucoup plus d'états que nécessaire. Bien que notre exemple n'est pas forcément représentatif puisqu'il correspond au pire cas imaginable, il illustre cependant le fait que le *proviso* $C2^{L}_{s}$ n'est pas adapté pour certains types de modèles.

Le *proviso* basé sur la notion de *sticky transition* [48] pourrait résoudre ce problème pour peu que les transitions *sticky* soient choisies correctement, e.g., transitions t_1, t_2, t_3 et t_4 dans notre exemple. Cependant, l'analyse statique permettant la détection des *sticky transitions* n'est pas aisée et dépend en plus du formalisme. Dans le cas d'un formalisme comme Promela, où la notion de processus est pourtant explicite, les mécanismes de détection peuvent aboutir à des fausses détections de *sticky transitions*. Dans le cas des réseaux de Petri, comme nous l'avons vu lors des stratégies de partitionnement, la notion de processus étant encore moins évidente, le risque serait alors d'obtenir de nombreuses fausses détections et d'aboutir ainsi à des coûteuses approximations pour lesquelles un grand nombre de transitions seraient considérées comme *sticky*.

L'algorithme *Twophase* [49] peut également aboutir à une réduction optimale sur cet exemple. Cependant, cet algorithme est basé sur un principe – toujours sélectionner un singleton – qui peut, pour certains modèles, s'avérer beaucoup trop restrictif et par là même, inefficace. Par exemple, ses performances sont particulièrement limitée lorsque des processus peuvent agir de façon indéterministe. De plus, il est incompatible avec certaines techniques de raffinement de la relation de dépendance, e.g., [36].

Nos objectifs étaient donc les suivants : définir un *proviso* qui (1) puissent être une alternative efficace lorsque les autres *proviso* n'arrivent pas à réduire efficacement l'espace d'état et (2) ne soit pas lié à un formalisme particulier et puisse être implémenté dans n'importe quel model checker.

Nous présenterons donc dans un premier temps un *proviso* alternatif pour les propriété de sûreté qui propose une approche beaucoup moins défensive que les *proviso* classiques et qui permet d'aboutir à une réduction efficace de l'espace d'état. Nous présenterons ensuite une version alternative du *proviso* pour les propriétés de vivacité qui se comporte souvent mieux que le *proviso* $C2^{L}_{s}$.

6.2.2 Un proviso pour les propriétés de sûreté

Nous proposons ici une nouvelle version du *proviso* pour les propriétés de sûreté, basé sur un algorithme de recherche en profondeur. Il effectue toujours une recherche dans la pile afin d'éviter

```
dfs (s)
1   H ← H ∪ {s}
2   s.expanded ← expanded
3   s.inStack ← true
4   let P be a persistent set that
5     satisfies C2e^S(s, P) or en(s)
6     if there is no such set
7     if P = en(s) then
8       expanded ← expanded + 1
9     end if
10    for a ∈ P do
11      let s -a-> s'
12      if s' ∉ H then
13        dfs(s')
14      end if
15    end for
16    if P = en(s) then
17      expanded ← expanded − 1
18    end if
19    s.inStack ← false
```

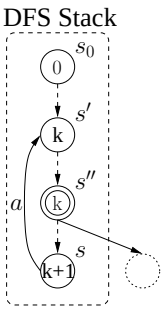
```
C2e^S (s, P)
1   for a ∈ P do
2     let s -a-> s'
3     if
4       s' ∉ H or
5       ¬s'.inStack or
6       s.expanded > s'.expanded then
7       return true
8     end if
9   end for
10  return false
```

```
search ()
1   H ← ∅
2   expanded ← 0
3   dfs(s0)
```

FIG. 6.8 – Algorithme de recherche en profondeur utilisant le *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$

un retardement infini d'actions mais relâche considérablement les conditions pour lesquelles une transition est acceptée.

Le pseudo-code de la figure 6.8 présente un algorithme d'ordre partiel basé sur ce *proviso* alternatif $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$. Le principe est extrêmement simple ; il consiste à associer un entier $s.expanded$ à *chaque état s de la pile*. Cet entier stocke simplement le nombre d'état complètement étendus présents dans la pile *sous* s, i.e., entre s_0 et s. La variable globale *expanded* conserve la valeur de ce nombre d'états complètement étendus. Lorsqu'une action a issue de s aboutit à un état s' présent dans la pile, on compare alors le nombre d'état complètement étendus dans la pile, i.e. la valeur de $s.expanded$, au nombre associé à s', i.e., $s'.expanded$. Si le premier est strictement supérieur cela signifie alors qu'il existe un état s'' complètement étendu dans la pile entre s et s'. On sait alors que s'' est accessible pour s et qu'ainsi, les actions franchissables pour s seront nécessairement exécutées sur le chemin allant de s à s''. Ce mécanisme est illustré par la figure ci-contre. La valeur associée à chaque état correspond à la valeur de son attribut *expanded*.



Ce *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$ est clairement meilleur que $\mathrm{C2}^{\mathrm{S}}_{\mathrm{s}}$ dans le sens où il calculera toujours de plus petits ensembles persistants (mais pas nécessairement de plus petits espaces d'états). Il peut être vu comme une optimisation de $\mathrm{C2}^{\mathrm{S}}_{\mathrm{s}}$: en supprimant l'attribut *expanded* associé à chaque état de la pile et en changeant la condition de la fonction $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$, on retrouve le *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{s}}$.

Le prix à payer est en un léger sûrcout mémoire nécessaire pour l'attribut *expanded* de chaque état de la pile (en général cette valeur est stockée dans un entier stocké sur 32 bits). Notons toutefois que cet attribut n'est plus nécessaire une fois que l'état est ôté de la pile et ne sera donc pas conservé. Nous verrons également dans les résultats expérimentaux que ce léger sûrcout est largement compensé par les réductions obtenues.

Pour démontrer la validité du *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$, nous allons prouver que la fonction de réduction possède un témoin (*witness*) [65].

Définition 6.7 (Fonction témoin (Witness function))**.** *Soit* $\mathcal{T} = (S, s_0, A, \rightarrow)$ *un STG, r une*

fonction de réduction de $\mathcal{T}$ *et* $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ *la réduction de* $\mathcal{T}$ *par* r*. Un mapping* $W : S_r \rightarrow \mathbb{N}$ *est un témoin pour* r *ssi :*

$$\forall s \in S_r, r(s) \neq en(s) \Rightarrow \exists (s, a, s') \in \rightarrow_r \text{ such that } W(s') < W(s)$$

L'intuition derrière l'idée de fonction témoin est que pour chaque état s partiellement étendu du graphe réduit, on peut trouver un successeur s' de s avec $W(s') < W(s)$ auquel on peut déléguer l'exécution des actions ignorées pour s. En ré-itérant sur s' on obtient une séquence $W(s), W(s'), \ldots$ de nombres décroissants. Comme l'espace d'état est fini, on trouvera alors forcément un état s'' tel que $W(s'') \geq W(s''')$ pour chacun de ses successeurs s'''. Pour un tel état, on a alors $r(s'') = en(s'')$ et ainsi, toutes les actions ignorées en s qui n'ont pas été explorées sur le chemin de s à s'' appartiennent à $r(s'')$. On peut alors en déduire que le graphe réduit possède un témoin [65].

Lemme 6.1. *Le proviso* $C2_e^S$ *implique le proviso* $C2^S$*.*

Démonstration. Soit $W : S_r \rightarrow \mathbb{N}$ une fonction qui énumère les états du graphe réduit $(S_r, s_{0r}, A_r, \rightarrow_r)$ dans l'ordre dans lequel ils sont supprimés de la pile : s_0 est mappé vers $|S_r| - 1$ tandis que le premier état à être dépilé est mappé vers 0. Soit F_W un ensemble d'état de S_r qui viole les conditions témoins, i.e., définies par :

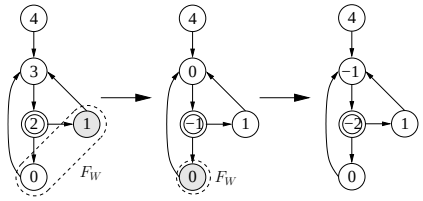
$$F_W = \{s \in S_r \mid r(s) \neq en(s) \wedge \forall (s, a, s') \in \rightarrow_r, W(s') \geq W(s)\}$$

Observons alors l'algorithme lorsqu'il traite un état $s \in F_W$. Chaque successeur $s' \in S_r$ de s est tel que $s' \in H \wedge s'.inStack$. Dans le cas contraire, s' quitte la pile avant s et $W(s') < W(s)$ ($\Rightarrow s \notin F_W$). De plus, il doit y avoir un état s' tel que $s \xrightarrow{a}_r s'$ pour tout $a \in r(s)$ et $s'.expanded < s.expanded$. Sinon, $r(s) = en(s)$ ($\Rightarrow s \notin F_W$).
Il existe alors un chemin $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n$ tel que $s' = s_1$, $s_1.inStack \wedge \cdots \wedge s_n.inStack$ et $r(s_n) = en(s_n)$. On peut alors définir une nouvelle fonction W' telle que

1 – $W'(s_1) < W(s)$ and $W'(s_1) < W(s_1)$
2 – $\forall s_i \in \{s_2, \ldots, s_n\}, W'(s_i) < W'(s_{i-1})$ and $W'(s_i) < W(s_i)$
3 – $\forall s \notin \{s_1, \ldots, s_n\}, W'(s) = W(s)$

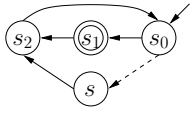
Comparons alors $F_{W'}$ et F_W. Le point 1 implique que $s \notin F_{W'}$. De plus, il est évident, d'après les 3 points, que $F_{W'} \setminus F_W = \emptyset$, i.e., W' n'introduit pas de nouvel "état invalide". On a alors $|F_{W'}| < |F_W|$.
En réitérant la même opération sur W' jusqu'à ce que $F_W = \emptyset$ on obtient alors un témoin W. □

Les différentes étapes de la construction d'une fonction témoin sont illustrées à l'aide de la figure ci-contre. Les états sont numérotés à l'aide de la fonction W. A chaque étape, l'état gris correspond à l'état s de la profondeur qui viole les conditions de la fonction témoin.



6.2.3 Un proviso pour les propriétés de vivacité

Les conditions qui garantissent une réduction correcte doivent être renforcées si l'on souhaite vérifier la validité de propriété de vivacité, e.g., des propriétés LTL-X. La réduction doit en effet assurer que pour chaque cycle dans l'espace d'état, toute transition franchissable pour un état du cycle sera franchi à partir d'un état du cycle. Nous avons vu précédemment qu'une condition suffisante consiste à étendre totalement un état sur chaque cycle du graphe.
Notre souhait était alors d'adapter l'idée du *proviso* $C2_e^S$, que nous venons de présenter, pour la vérification de propriétés de vivacité. Malheureusement, une utilisation directe de ce *proviso* n'aboutit pas au comportement souhaité. Nous allons illustrer le problème à l'aide du simple graphe présenté ci-dessous.



Supposons que notre algorithme commence par explorer l'état s_0, puis passe par s_1 qui est complètement étendu, avant d'atteindre finalement s_2. Puisque s_1 est dans la pile entre s_0 et s_2 et qu'il est complètement étendu, l'ensemble persistent réduit à la seule action allant de s_2 à s_0 est considérée comme valide.
Supposons alors que l'algorithme, lors de la phase de dépilement, atteigne à nouveau s_0, et exécute alors un séquence $s_0 \rightarrow \cdots \rightarrow s$ telle qu'*aucun état de cette séquence ne soit complètement étendu*. Si l'on utilise le *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$, alors le singleton $\{s \rightarrow s_2\}$ est un ensemble persistent valide. Nous aboutissons alors à la fermeture d'un cycle qui ne contient aucun état totalement étendu et dans lequel une action pourra être ignorée : $s_0 \rightarrow \cdots \rightarrow s \rightarrow s_2 \rightarrow s_0$.
Pour éviter des telles situations, il est nécessaire d'effectuer des tests supplémentaires. En particulier, nous allons interdire la possibilité pour s d'atteindre s_2 sans être totalement étendu.

Le pseudo-code de notre algorithme est présenté à la figure 6.9. En plus de l'attribut *expanded* du *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$, notre nouveau *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}}$, que nous appelons *proviso coloré*, associe une information de couleur pour chaque état. Les états peuvent alors être coloré en vert, orange ou rouge. Ces couleurs nous donne des informations cruciales lorsque l'on souhaite déterminer si une action est autorisée ou par (fonction $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}}$).

green les états verts sont des états *sûrs*. Ils peuvent alors être atteints par n'importe quel autre état sans risque de fermer un cycle invalide. Intuitivement, les états verts sont des états qui sont soit complètement étendus, soit des états dont tous les successeurs soit totalement verts.

red les états rouges sont des états dangereux. Un état ne peut atteindre un état rouge sans avoir été totalement étendu. Cela pourrait fermer un cycle invalide comme dans l'exemple présenté précédemment. Les états rouges sont des états qui ne sont plus dans la pile de recherche.

orange les états oranges sont des états potentiellement dangereux. Un état orange est un état de la pile qui peut être atteint par un autre état sous la condition du *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{e}}$: un état complètement étendu appartient au cycle refermé dans la pile.

Les couleurs sont alors attribuées aux états de la façon suivante.
Lorsqu'un nouvel état est généré et empilé dans la pile de recherche, il est coloré en vert s'il est totalement étendu et orange sinon. La couleur orange est attribuée dans la fonction push_state avant le calcul de l'ensemble persistent P pour résoudre le cas où P contient une transition aboutissantt à l'état à partir duquel elle est franchi. Les états oranges sont donc tous les états partiellement étendus de la pile.
Un état orange qui quitte la pile est coloré en vert si tous ses successeurs sont verts et coloré en rouge sinon. Les couleurs rouge et vert sont donc des couleurs finales, i.e. un état rouge ou vert ne peut changer de couleur, tandis que la couleur orange est une couleur transitoire avant de devenir vert ou rouge.

Les lignes 13-18 de la procédure dfs ont pour but de traiter les situations pour lesquelles l'état s est partiellement étendu et atteint un état rouge s' qui n'était pas dans H lorsque l'ensemble persistent de s a été calculé. Il faut alors étendre complètement s, lui assigner la couleur vert et recommencer l'exploration à partir de s. Dans la pratique, cette situation se présente assez rarement.

Reprenons alors notre exemple précédent et regardons comment notre algorithme va fonctionner. Lorsque l'état s_2 est dépilé, il est coloré en rouge puisque son unique successeur, l'état s_0, est orange, i.e. partiellement étendu et dans la pile. On continue alors de dépiler les états jusqu'à s_0 pour atteindre ensuite s. Puisque s_2 est un état rouge, toute action aboutissant à s_2 partir de s est interdite si s n'est pas totalement étendu. Il faut alors choisir un autre ensemble persistent ou étendre complètement s.

Pour prouver la validité de notre *proviso*, nous procédons en deux étapes. Nous allons commencer par montrer que le graphe réduit ne peut contenir un cycle d'états rouges.

```
dfs (s)
1   H ← H ∪ {s}
2   push_state(s)
3   let P be a persistent set that
4   satisfies C2c^L(s, P) or en(s)
5   if there is no such set
6   if P = en(s) then
7      expanded ← expanded + 1
8      s.color ← green
9   end if
10 search_loop:
11  for a ∈ P do
12     let s -a-> s'
13     if s' ∉ H then
14        dfs(s')
15     else if s.color = orange and s'.color = red then
16        s.color ← green
17        P ← en(s)
18        goto search_loop
19     end if
20  end if
21  if P = en(s) then
22     expanded ← expanded − 1
23  end if
24  pop_state(s)

search ()
1   H ← ∅ ; expanded ← 0 ; dfs(s0)
```

```
push_state (s)
1   s.inStack ← true
2   s.color ← orange
3   s.expanded ← expanded

pop_state (s)
1   s.inStack ← false
2   if s.color = orange then
3      if ∀a ∈ r(s), s -a-> s', s'.color = green then
4         s.color ← green
5      else
6         s.color ← red
7      end if
8   end if

C2c^L (s, P)
1   for a ∈ P do
2      let s -a-> s'
3      if s' ∈ H and
4      (s'.color = red or
5      (s'.color = orange and
6      s'.expanded = s.expanded)) then
7         return false
8      end if
9   end for
10  return true
```

FIG. 6.9 – Un algorithme de recherche en profondeur implémentant notre *proviso* pour les propriétés de vivacité.

Proposition 6.1. *Soit $\mathcal{T} = (S, s_0, A, \rightarrow)$ un STG et $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ une réduction obtenue en utilisant l'algorithme présenté à la figure 6.9. Alors, il n'y a aucun cycle d'états rouges dans $\mathcal{T}_r$, i.e., $\forall s_1, \ldots, s_n \in S_r$,*

$$s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1 \Rightarrow \exists i \in \{1..n\} \mid s_i.color = green$$

Démonstration. Supposons qu'il existe un cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$ avec $s_i.color = red, \forall i \in [1..n]$ et tel que s_1 est le premier état visité par l'algorithme, i.e., empilé sur la pile de recherche.
Nécessairement, pendant la recherche on a atteint une configuration pour laquelle

1. les états $s_1, \ldots, s_i$ sont sur le haut de la pile.
2. $s_1.color = \cdots = s_i.color = orange$.
3. il existe $a \in r(s_i)$ tel que $s_i \xrightarrow{a} s_j$ et $s_j \in H$.

Observons alors cette configuration. Par hypothèse, $s_j.color \neq green$, donc, $s_j.color \in \{orange, red\}$. Considérons alors ces deux hypothèses.

$s_j.color = red$ ($\Rightarrow s_j$ a quitté la pile)
Là encore, on considère deux cas.

$s_j \in H$ lorsque $r(s_i)$ est calculé
Nécessairement, $s_j.color = red$ lorsque $r(s_i)$ est calculé. Sinon, s_j est au dessus de s_i dans la pile de parcours et $s_j.color = orange$ lorsqu'on atteint s_j à partir s_i. D'après la condition du *if* de la ligne 3 de $\mathrm{C2}_c^{\mathrm{L}}$, $s_j \in H \wedge s_j.color = red \Rightarrow r(s_i) = en(s_i)$, et alors $s_i.color = green$ après l'affectation effectuée à la ligne 8 de dfs.

$s_j \notin H$ lorsque $r(s_i)$ est calculé
Alors, lorsque s_j est atteint à la ligne 11 de la fonction dfs on sait, par hypothèse, que $s_j \in H$, $s_j.color = red$ et $s_i.color = orange$. Alors s_i est coloré en vert à la ligne 16.

$s_j.color = orange$ ($\Rightarrow s_j$ est sûr la pile)
L'état s_j a été empilé avant s_i. Ainsi, on a $s_j.color = orange$ lorsque $r(s_i)$ a été calculé. A partir de la fonction $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}}$, si $s_j \in H \wedge s_j.color = orange$ alors $s_j.expanded < s_i.expanded$. Sinon, on aurait $r(s_i) = en(s_i)$ et s_i serait alors coloré en vert à la ligne 8 de la fonction dfs. Puisque $s_j.expanded < s_i.expanded$, il existe s_k avec $j < k < i$ tel que $r(s_k) = en(s_k)$. Par conséquent, $s_k.color = green$ à la ligne 8 de la fonction dfs.

Dans le deux cas, on a bien un état dans le cycle. □

Nous allons désormais prouver que si un cycle du STG réduit contient un état vert, alors il contient un état totalement étendu.

Proposition 6.2. *Soit $\mathcal{T} = (S, s_0, A, \rightarrow)$ un STG et $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ sa réduction obtenue en utilisant l'algorithme de la figure 6.9. Dans chaque cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$, s'il existe un état s_i tel que $s_i.color = green$ alors il existe un état s_j tel que $r(s_j) = en(s_j)$.*

Démonstration. Considérons, dans cette preuve, un cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$ tel que $s_i.color = green$ pour tout $i \in \{1..n\}$.
Supposons tout d'abord qu'il y a un état rouge dans le cycle. S'il existe un état s_i tel que $s_i.color = red$ alors, nécessairement, il existe deux états s_j et s_k tels que $s_j.color = green$, $s_k.color = red$ et $s_j \rightarrow_r s_k$ (sinon, le cycle ne contiendrait que des états rouges). Puisqu'il est évident qu'un état vert avec un successeur rouge est totalement étendu, la propriété est vérifiée pour ce premier cas.
Supposons désormais que $\forall i \in \{1..n\}$, $s_i.color = green$. Nécessairement, pendant la recherche, un état s_i a atteint un état s_j de la pile. Puisque $s_j.inStack = true$ alors $s_j.color \in \{orange, green\}$. Considérons alors les 2 possibilités suivantes.

$s_j.color = green$ **-** On a alors, pour chaque état s coloré en vert de la pile, $r(s) = en(s)$.

$s_j.color = orange$ **-** Lorsque s_i quitte la pile (avant s_j) il devient rouge puisqu'il possède un successeur non vert. Ceci est alors incohérent avec notre hypothèse initiale selon laquelle tous les états du cycle sont verts.

Dans les deux cas, il y a bien un état totalement étendu dans le cycle. □

Il est alors évident de prouver la validité de notre *proviso* pour les propriétés de vivacité.

Lemme 6.2. *Le* proviso $C2^L_c$ *implique le* proviso $C2^L$.

Démonstration. Ce lemme est une conséquence directe des propositions 6.1 et 6.2. □

a. Anticipation de la phase de dépilement

La couleur rouge apparaît lorsqu'un état partiellement étendu s atteint un état orange. Ainsi, lorsque s est retiré de la pile, il devient rouge et cette couleur est propagée à tous ses prédécesseurs dans la pile. Cette façon de faire est extrêmement défensive puisque tous les états oranges qui atteindront s seront alors colorés en rouge. Dans certaines situations, il est toutefois possible de coloré directement des états oranges en vert en anticipant la phase de dépilement.
Nous illustrons le principe de cette optimisation à l'aide de la figure 6.10. Les lettres G, R et O correspondent aux couleurs *green*, *red* et *orange*. Si l'on considère l'algorithme de base, lorsque l'état s est exploré, il atteint l'état s' et devient alors rouge lorsqu'il est dépilé. Cependant, comme tous les arcs sortants de s' ont été visités et que son unique successeur est vert, on sait qu'il deviendra vert lorsqu'il quittera la pile. On peut alors immédiatement colorer s' en vert. La conséquence directe de cette optimisation est que l'état s atteindra uniquement des états verts et sera donc coloré en vert.

L'implémentation de cette optimisation requiert l'utilisation d'une variable booléenne supplémentaire pour chaque état de la pile qui permet de savoir si tous les arcs sortants de l'état ont été visités. Nous introduisons également une nouvelle couleur : le violet (*purple*). L'unique intérêt de

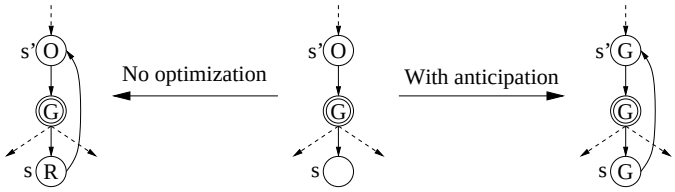


FIG. 6.10 – Illustration de l'optimisation

cette nouvelle couleur est de faciliter l'implémentation de cette optimisation : les états violets sont traités comme des états oranges lorsque l'on vérifie le *proviso*.
En utilisant ce *proviso* optimisé, noté $C2^{L}_{c\star}$, l'algorithme fonctionne de la façon suivante.
Lorsque la couleur verte est assignée à l'état courant ou lorsqu'une action aboutit à un état vert, la pile est scannée à partir du haut de la pile jusqu'à rencontrer un état vert ou violet ou un état orange pour lequel des arcs sortants n'ont pas été visités. Tous ces états sont alors colorés en vert. De façon similaire, lorsqu'une action abouti à l'exploration d'un état orange ou violet, l'algorithme scanne la pile jusqu'à rencontrer un état vert ou un état violet et colore tous ces états en violet.

Cette optimisation peut être particulièrement intéressante puisque la condition C1 aboutit régulièrement à la création de singletons, e.g, constitué d'une transition unique qui ne modifie qu'une variable locale, ou a des états totalement étendus. Dans ces situations, l'optimisation se révèle assez efficace puisqu'elle permet de colorer en vert la plupart des états de la pile : dès qu'un état totalement étendu est rencontré, la couleur verte est propagée de haut au bas de la pile pour tous les états.

Alors qu'il est évident que le *proviso* que nous avons présenté pour les propriétés de sûreté est toujours plus efficace que le *proviso* classique, notre *proviso* coloré n'est pas plus efficace que le *proviso* classique pour les propriétés de vivacité dans tous les cas.

Les *proviso* $C2^{L}_{s}$ et $C2^{L}_{c}$ sont tous les deux basés sur la notion d'états dangereux ou sûrs. Avec le *proviso* $C2^{L}_{s}$, les états considérés comme dangereux sont tous les états de la pile (ou plus généralement, tous les états fermés [37]) tandis que, dans le cas du *proviso* coloré, les états dangereux peuvent être des états qui ne sont plus dans la pile. Il est alors important d'expérimenter ces *provisos* pour observer leurs comportement dans la pratique.

6.2.4 Evaluations

Nous avons implémenté les algorithmes proposés dans notre model checker Helena [111] qui permet la vérification de propriétés d'accessibilité ou la détection d'états morts dans des réseaux de Petri colorés de haut-niveau. Nous avons également implémenté les *provisos* classiques utilisés notamment par Spin pour les comparer aux notres.
Nous avons observé, comme c'est le cas dans [38], que les algorithmes basés sur un parcours en largeur ont tendance à être moins efficaces que ceux basés sur un parcours en profondeur. En effet, sur les différents modèles analysés, seul un (le protocole *slotted ring*) donne de meilleures réductions lors du parcours en largeur. Hors les différences avec une recherche en profondeur sont, dans ce cas là, quasiment insignifiantes. C'est pourquoi nous ne les reportons pas ici et que nous préférons nous focaliser sur les comparaisons entre les *provisos* utilisés par des parcours en profondeur.
Les résultats de ces expérimentations sont reportés dans le tableau 6.1. A chaque fois, nous avons effectué plusieurs explorations : sans aucune méthode de réduction (colonne No POR), sans vérification du phénomène d'ignorance (colonne PS), avec les *provisos* pour les propriétés de sûreté (colonnes $C2^{S}_{s}$ et $C2^{S}_{e}$) et aves les *provisos* pour les propriétés de vivacité (colonnes $C2^{L}_{s}$, $C2^{L}_{c}$ et $C2^{L}_{c\star}$). Les valeurs contenues dans les colonnes No POR et PS doivent alors être considérées comme des bornes respectivement supérieures et inférieures lors de la comparaison des *provisos*.
Pour chaque exécution, nous avons reporté le nombre d'états explorés dans le graphe réduit ainsi

que la quantité de mémoire nécessaire pour stocker ces états en mémoire. Dans certains cas, la mémoire disponible ne permettait pas l'exploration du graphe. Ces cas sont indiqués par "oom" (*out of memory*).

Pour les propriétés de sûreté, une comparaison des colonnes PS et $C2^S_e$ montre que notre *proviso* aboutit à d'excellentes réductions. Pour huit modèles il n'introduit aucun état non visité par un algorithme n'effectuant pas de détection du phénomène d'ignorance. Pour l'algorithme de Lamport, il aboutit à l'exploration de quelques milliers d'états supplémentaires qui sont assez peu comparé à la taille de l'espace d'état de ce modèle. Il double également la taille du graphe pour le modèle du système d'allocation de ressources. Dans ce modèle, un processus peut potentiellement diverger et effectuer une séquence infinie dans laquelle aucune synchronisation n'est effectuée. Il y a donc un risque potentiel d'ignorer des actions, il est donc naturel que notre *proviso* ajoute quelques états supplémentaires. Néanmoins, notre *proviso* se comporte beaucoup mieux que $C2^S_s$ sur ce modèle. Ces résultats confirment notre intuition initiale : une DFS ferme rarement un cycle qui ne contient

No POR	PS	PS + Safety proviso		PS + Liveness proviso		
		$C2^S_s$	$C2^S_e$	$C2^L_s$	$C2^L_c$	$C2^L_{c\star}$
Load-balancing system (7 clients, 3 servers)						
1 574 530	72 093	631 056	72 093	630 997	211 012	72 194
26.4 MB	1.2 MB	10.7 MB	1.5 MB	10.7 MB	4 MB	1.3 MB
A peer-to-peer communication protocol (8 processes)						
743 580	163	72 852	163	72 852	884 830	252 315
12.1 MB	0.1 MB	1.2 MB	0.1 MB	1.2 MB	15.6 MB	5.2 MB
Resource allocation system (4 processes)						
2 550 759	72 637	1 449 206	151 531	1 783 881	754 878	607 004
49.9 MB	1.5 MB	28.7 MB	3.6 MB	35.2 MB	23.2 MB	15.6 MB
Lamport's mutual exclusion algorithm (4 processes)						
1 914 784	1 052 518	1 282 950	1 055 985	1 455 606	1 304 311	1 304 310
41.02 MB	22.5 MB	27.4 MB	26.7 MB	31.3 MB	31.6 MB	31.6 MB
Peterson's mutual exclusion algorithm (4 processes)						
3 407 946	259 942	356 068	259 942	356 698	292 622	260 608
49.3 MB	3.7 MB	5.1 MB	4.7 MB	5.1 MB	4.8 MB	4.3 MB
Production cell (8 plates)						
oom	396 931	1 024 422	396 931	1 138 954	495 543	451 355
	18.2 MB	46.3 MB	19.1 MB	51.4 MB	24.2 MB	21.9 MB
Slotted ring protocol (7 processes)						
439 296	287 508	413 321	287 508	437 579	401 803	304 417
6.1 MB	4 MB	5.8 MB	5.1 MB	6.1 MB	6.5 MB	4.9 MB
The chameneos (4 tasks)						
oom	415 361	899 295	415 361	899 295	733 654	494 123
	4.7 MB	10.4 MB	6.4 MB	10.4 MB	10.2 MB	6.9 MB
The dining philosophers (6 tasks)						
10 888 070	109 222	174 354	109 222	174 354	115 333	110 190
136 MB	1.3 MB	2.1 MB	1.7 MB	2.1 MB	1.7 MB	1.6 MB
A client-server program (4 clients, 2 servers)						
oom	87 129	99 430	87 129	99 430	159 202	108 659
	1.4 MB	1.6 MB	1.7 MB	1.6 MB	2.8 MB	1.9 MB

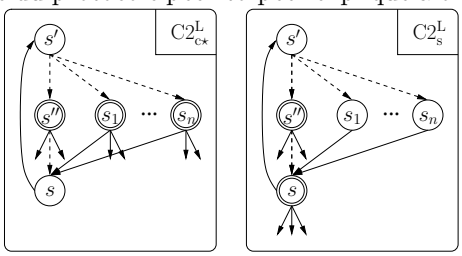
Tab. 6.1 – Comparaison des différents *provisos* implémentés dans Helena.

pas d'état totalement étendu. Dans les systèmes concurrents, il y a généralement plusieurs points de synchronisation, e.g, accès à des variables partagées, acquisition d'un verrou, ... Lorsqu'un processus atteint ces points, il est probable que l'algorithme étende complètement l'état. Il nous semble qu'un point faible du *proviso* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{s}}$ est qu'il n'exploite pas d'informations sur le passé de la recherche que la pile peut nous fournir. Notre *proviso* est alors quasiment optimal puisqu'il interdit uniquement à l'algorithme de fermer un cycle pour lequel aucun état n'a été complètement étendu.

On observe alors également que les *provisos* $\mathrm{C2}^{\mathrm{S}}_{\mathrm{s}}$ et $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ augmentent parfois brutalement la taille du graphe. Cela confirme notre intuition selon laquelle ces *provisos* ne sont pas adaptés à certains types de systèmes. On peut alors trouver plusieurs modèles pour lesquels ces *provisos* aboutissent à l'exploration de beaucoup plus d'états que nécessaire. Pour certains modèles, e.g., le *slotted ring protocol* ou le système d'allocation de ressource, la colonne No POR montre qu'ils n'effectuent quasiment aucune réduction ou des réductions extrêmement faibles.

En comparant les colonnes PS et $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ on peut évaluer notre *proviso* en terme d'états introduits. Les résultats sont plutôt convaincants. Pour sept modèles sur dix, les réductions obtenues sont similaires. Pour le protocole peer-to-peer et l'allocateur de ressources, l'introduction de ces états aboutit toutefois à une réelle augmentation de la taille du graphe. Comme nous l'avons précisé précédemment, ceci n'est pas étonnant dans le cas de l'allocateur de ressources. Pour ce qui est du protocole peer-to-peer, nous verrons que notre *proviso* n'est pas adapté à la structure de son graphe.

Dans l'ensemble, $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ aboutit à de meilleures réductions que $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$. Pour certains modèles la différence est assez impressionnante. Nous pouvons citer par exemple le système d'équilibrage de charge. Pour certains autres modèles, e.g., l'algorithme de Lamport, la différence est nettement plus limitée. Nous n'avons cependant que deux modèles sur les dix pour lesquels $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ se comporte mieux : le programme client-serveur et le protocole peer-to-peer. Pour le premier, la différence est cependant assez faible. Une observation plus fine de la structure du graphe du protocole peer-to-peer explique alors les mauvais résultats obtenus avec $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ comparé à $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$. Nous avons observé que la situation illustrée sur la figure ci-contre apparaît fréquemment. Avec le *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$, lorsque s est exploré, il peut être étendu partiellement puisque l'état totalement étendu s'' est entre s et s' dans la pile. Plus tard, lorsque les états s_1, ..., s_n sont atteints, l'algorithme les étend totalement puisque s est alors rouge. Dans le cas du *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$, s ne pourra atteindre s' sans avoir été totalement étendu. Les états s_1, ..., s_n peuvent alors être partiellement étendus puisqu'ils atteignent un état qui a quitté la pile. Ceci explique alors pourquoi, sur cet exemple, $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ étend totalement plus d'états que $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$.



Nous terminerons alors cette section avec quelques remarques sur la consommation mémoire. Nous remarquons que malgré le léger ajout d'information (lié à la sauvegarde de la couleur) pour chaque état, $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ est généralement plus efficace que $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$. Pour un seul model – l'algorithme de Lamport – $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ aboutit à de meilleures réductions que $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ mais nécessite plus de mémoire. Cependant, même dans ce cas, la différence est insignifiante.

6.3 Réductions d'ordres partiel en réparti

Comme nous l'avons expliqué précédemment, l'objectif de la parallélisation ne devait pas aboutir à l'abandon des autres techniques permettant de combattre le problème de l'explosion combinatoire. Nous avons vu que les méthodes de réductions structurelles n'ont pas besoin d'être adaptées à la parallélisation puisqu'elles sont effectuées de façon statique, donc avant l'exploration de l'espace d'état.

Nous avons ensuite vu comment adapter les techniques de représentation d'états à des environnements répartis.
Dans cette section, nous allons voir comment adapter les méthodes d'ordre partiel lorsque l'exploration s'effectue en parallèle.

6.3.1 Problématique

Comme nous l'avons vu précédemment, les techniques d'ordre partiel utilise des contraintes pour construire le graphe réduit à la volée. La contrainte C1 pouvant être vérifiée statiquement, elle n'est pas problématique dans le cas d'une exploration effectuée en parallèle. Il en est de même pour la condition C0 qui peut être vérifiée localement puisqu'elle ne fait intervenir que l'état lui-même et son ensemble persistent.

Considérons maintenant la condition $C2^L$. Pour permettre son utilisation, nous avons vu qu'elle est souvent transformé en une condition qui fait intervenir les structures de données utilisées par le parcours. Dans le cas d'une exploration effectuée en profondeur, on utilise généralement le *proviso* $C2^L_s$. Ce *proviso* se base sur l'utilisation de la pile de parcours. Ceci pose un premier problème puisque, dans le cas d'une exploration effectuée en parallèle, on ne conserve pas la pile de parcours globale mais uniquement un pile de parcours locale. Si l'on considère l'espace d'état présenté à la figure 6.11, lorsque l'arc 5 est exploré, une nouvelle pile est gérée par le processus responsable de l'état s_3. Lorsque ce processus explore l'arc 4, s_1 n'est alors pas dans sa pile ! On ferme alors un cycle dans lequel aucun état n'est complètement étendu.

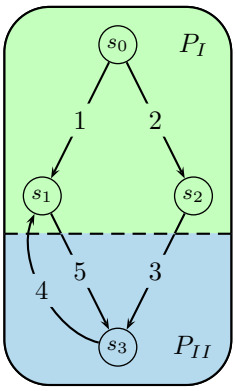


Fig. 6.11 – Problématique du parcours en parallèle

Considérons maintenant ce même espace d'état en supposant qu'à chaque fois, on conserve la pile de parcours et que les valeurs sur les arcs indiquent leur ordre d'exploration.
En partant de l'état s_0, l'arc 1 est franchi et aboutit à l'exploration de l'état s_1. Le successeur de s_1 n'étant pas sur la partition P_I, il est envoyé au bon processus. L'état s_1 est alors dépilé puis s_2 est exploré. Là encore, le successeur n'appartenant pas à P_I, il est envoyé à processus en charge de son exploration. Supposons alors que ce dernier message soit reçu en premier par le processus responsable de l'exploration de s_3. La pile reçue est constituée de $s_0 \rightarrow s_2$. Le successeur de s_3 devant être exploré par le site I, il lui est envoyé. Lorsque le site I reçoit l'état s_1 avec la pile $s_0 \rightarrow s_2 \rightarrow s_3$, l'état s_1 étant considéré comme visité et n'étant pas présent dans la pile, l'exploration s'arrête sans étendre ni s_1, ni s_3. De même lorsque le site II reçoit l'état s_3 envoyé initialement avec la pile $s_0 \rightarrow s_1$, s_3 étant considéré comme visité et n'appartenant pas à la pile, l'état n'est pas totalement étendu. Cette exploration aboutit donc bien à un cycle dans le graphe dans lequel aucun état totalement étendu n'apparaît malgré l'envoi des piles de parcours.
Ce comportement provient du fait que la pile d'exploration seule ne suffit pas ici puisque le parcours n'est plus globalement cohérent. Le *proviso* $C2^L_s$ ne peut donc plus être appliqué tel quel.

Plus exactement, il ne peut plus être appliqué sans modification dans le cas où des cycles faisant intervenir des états explorés par différents processus peuvent apparaître. Tant que l'exploration s'effectue localement, le *proviso* reste valide car la cohérence est conservée localement.

La cohérence dont nous parlons ici fait référence à la notion de cohérence évoquée dans la sous-section 2.1.6. L'algorithme séquentiel se base sur le fait que le parcours en profondeur explorera la séquence $s_0 \rightarrow s_1 \rightarrow s_3$ *avant* la l'exploration de la séquence $s_0 \rightarrow s_2 \rightarrow s_3$. Cette dépendance causale n'est plus garantie en répartie et rend donc l'algorithme séquentiel invalide lorsque les cycles sont répartis sur différents nœuds. Les solutions que nous présenterons ne cherchent pas à essayer de respecter les dépendances causales – ce qui serait assez coûteux – mais cherchent à autoriser un parallélisme total en essayant de gérer ces problèmes de non cohérence du parcours.

Lorsque l'exploration se fait de façon parallèle, il est donc nécessaire de gérer le cas où un cycle peut faire intervenir des états explorés par différents processus. Nous allons donc, dans un premier temps, présenter les solutions existantes. Puis nous présenterons les modifications à apporter à notre algorithme coloré qui, en relâchant les contraintes sur la pile de parcours, est à priori plus adapté à la répartition.

6.3.2 Etat de l'art

Quelques solutions ont déjà été proposées pour répondre au problème présenté précédemment. Toutes se basent sur des tests effectués uniquement sur la pile.

a. L'algorithme TwoPhase

L'algorithme *Twophase* est un algorithme de réduction d'ordre partiel assez original puisqu'il ne se base pas sur le *proviso* habituel qui effectue une recherche dans la pile.

Dans un premier temps nous allons présenter l'algorithme séquentiel proposé dans [49] puis son adaptation à un environnement réparti proposée dans [85].

i. Algorithme séquentiel

L'algorithme séquentiel *Twophase* est un algorithme plus simple que l'algorithme classique de Spin et peut, dans certains cas, obtenir de meilleures réductions.

L'algorithme procède de la façon suivante (cf. figure 6.12) : pour chaque état s_i, *Twophase* teste si un processus P_i possède un singleton *ample*. Si ça n'est pas le cas, l'état est complètement étendu (ie. tous ses successeurs sont explorés). Dans le cas contraire, supposons qu'il existe un singleton pour le processus P_i (transition t_i), alors *Twophase* explore l'état $s_j = t_i(s_i)$. Si s_j possède également un singleton pour un processus P_j (qui peut être P_i), alors l'algorithme reste dans la phase 1 et recommence pour s_j la procédure effectuée pour s_i. Lors de cette première phase, l'algorithme maintient une liste d'états visités par cette phase. Lorsqu'un état visité par cette phase est déjà présent dans la liste, l'algorithme passe au processus suivant (ligne `11`). Cette méthode permet donc d'éviter la recherche dans la pile de parcours.

Lorsqu'aucun singleton n'a été trouvé pour un processus, l'algorithme retourne dans la phase 2 qui consiste en une DFS classique. Tous les états visités lors de la première phase sont ajoutés à l'espace d'état.
Plus simplement, la phase 1 est une phase dans laquelle les réductions sont effectuées. Ce sont alors les états visités lors de cette phase qui sont dangereux puisqu'ils sont partiellement étendus. Il est alors nécessaire d'éviter qu'un état visité lors de cette phase ferme un cycle avec un autre état visité lors de cette même phase. En revanche, un cycle peut être fermé dans la phase 1 si l'on rencontre un état s_i visité lors de la phase 2 (car s_i est alors totalement étendu) ou si l'on rencontre

```
model_check ()
1   V_r ← ∅;  // Hash table
2   Twophase(s_0);

Twophase (s)
1   local list;
2   // Phase 1
3   (list, l) ← phase1(s);
4   // Phase 2 : classic DFS
5   if s ∉ V_r then
6      V_r ← V_r+ all state in list +s;
7      for each enabled transition t do
8         if t(s) ∉ V_r then
9            Twophase(t(s));
10        end if
11     end for
12  else
13     V_r ← V_r+ all state in list ;
14  end if
```

```
phase1 (in)
1   local olds, s, list;
2   s ← in;
3   list ← s;
4   for each process P do
5      while singleton_ampleset(s, P) do
6         // let t be the only enabled
7         // transition in P at s
8         olds ← s;
9         s ← t(olds);
10        if s ∈ list then
11           break;
12        end if
13        list ← list + s;
14     end while
15  end for
16  return (list, s);
```

FIG. 6.12 – Algorithme *Twophase* séquentiel pour les propriétés de vivacité.

un état s_j visité lors d'une autre phase 1, car dans ce cas, une phase 2 a été effectuée entre temps et des états totalement étendus sont alors présents dans le cycle.

ii. Twophase en environnement réparti

L'algorithme *Twophase* en réparti poursuit un mode de fonctionnement assez simple : la première phase s'effectue de la *même façon qu'en séquentiel* sans gérer les aspects répartis de la vérification. C'est uniquement lors de la seconde phase (une BFS répartie classique) qu'un état exploré est envoyé à son propriétaire si le processus courant n'est pas le propriétaire de cet état.

L'adaptation de l'algorithme *Twophase* à un environnement réparti est présenté à la figure 6.13.

Dans de nombreux cas, les états explorés par un nœud n_i lors de la première phase ne font pas partie de la (ou des) partition(s) dont il est responsable. Il est alors possible de marquer ces états afin qu'ils ne soient pas ajoutés à la partition de l'espace d'état stockée par n_i.

Les principales limitations de cet algorithme sont (1) les explorations potentiellement redondantes effectuées par la phase 1 puisqu'aucun contrôle sur la partition à laquelle appartiennent les états n'est effectuée et (2) l'utilisation parfois restrictive de singletons.

b. L'algorithme de Spin

Le premier algorithme présenté dans [80] proposait un algorithme de calcul d'ordre partiel réparti extrêmement pessimiste : chaque état exploré appartenant à un autre nœud est considéré comme étant dans la pile et entraîne donc une exploration complète de son prédécesseur. Le *proviso* ainsi modifié devient :

C2$^{\mathbf{L}}_{\mathbf{ds}}$ Si $r(s) \neq en(s)$ alors aucune action de $r(s)$ ne peut atteindre un état de la pile ni un état qui doit être exploré par un autre processus.

Dans le cas de partitions à faible degré de localité, les réductions obtenues par cette méthode peuvent alors être quasi nulles.

Le *proviso* proposé dans [84] est un raffinement de la solution précédente. C'est également une nouvelle solution aux limitations de l'algorithme *Twophase*. L'approche utilisée est, là aussi, basée

```
DistributedTwoPhase (myid)
1   s ← s0;
2   V ← ∅;  // Set of visited states
3   Q ← ∅;  // Queue of states waiting to be expanded
4   list ← ∅;  // The mini-list
5   s ← phase1(&list, s);
6   V ← V∪ states in the list;
7   i ← partition(s);
8   if i = myid then
9     Q ← Q ∪ s;
10  else
11    send(s, i);
12  end if
13  while search is not complete do
14    s ← head(Q);
15    t ← set of enabled transition in s;
16    for each transition in t do
17      if t(s) not in V then
18        s ← phase1(&list, t(s));
19        V ← V∪ states in the list;
20        if i = myid then
21          Q ← Q ∪ s;
22        else
23          send(s, i);
24        end if
25      end if
26    end for
27  end while
```

FIG. 6.13 – Algorithme *Twophase* réparti limité aux propriétés de sûreté.

sur le *proviso* classique avec recherche dans la pile.

Cette nouvelle condition se base sur l'observation que la partie de la pile à considérer est en fait la partie comprise entre l'état courant et le dernier état complètement étendu. En effet, une fois qu'un état a été complètement étendu, tous les cycles atteignant cet état à travers la pile contiennent cet état comme état complètement étendu.
L'idée étant ici de dire que la partie de la pile pertinente pour la détection de cycles ne contenant pas d'états totalement étendus est la partie comprise entre l'état courant et le dernier état totalement étendu, les auteurs utilisent, conjointement avec la pile de recherche, un tableau *history* de booléens de taille N, où N correspond au nombre de nœuds utilisés pour l'exploration. Pour chaque nœud visité par la DFS, le booléen correspondant est placé à *vrai*. Ainsi, lorsqu'un ensemble *ample* explore un état appartenant à un autre nœud, si la DFS n'est pas déjà passée par ce nœud, il n'y a aucun risque de fermer un cycle. L'état peut alors rester partiellement étendu. Le tableau de booléens peut alors être réinitialisé à chaque fois qu'un état est totalement étendu puisqu'il clôt une zone dangereuse de la pile.

Les auteurs utilisent alors une autre heuristique permettant de réduire les explorations complètes d'états en conservant dans l'espace d'état le tableau de booléens tel qu'il était lors de la visite de l'état. Ceci permet alors de réduire le nombre d'état totalement étendu en permettant à un état d'être exploré plusieurs fois.
On peut alors raffiner $\mathrm{C2}^{\mathrm{L}}_{\mathrm{ds}}$ en :

$\mathbf{C2}^{\mathbf{L}}_{\mathbf{ds}}$, Si un état s n'est pas complètement étendu, alors aucune transition de $ample(s)$ ne génère d'état présent dans la pile de recherche *locale* ni d'état appartenant à un autre nœud déjà visité par la DFS courante.

Deux scénario sont alors à considérer :

1. dans le cas où $ample(s)$ est un sous-ensemble de $enabled(s)$,
   - pour chaque transition allant vers un état s' tel que $owner(s') \neq owner(s)$, une copie $history'$ du tableau de booléens $history$ est créée dans laquelle $history'[owner(s)]$ est mis à vrai. On envoie ensuite le tuple $s', history'$ au propriétaire de s'
   - la DFS continue ensuite à partir de s en ne considérant que les états s' tels que s et s' appartiennent au même nœud. Si l'état s' a déjà été exploré et que son tableau $history'$ vérifie la condition suivante : $\forall i \in N, history[i] \Rightarrow history'[i]$, alors s' est considéré comme visité. Sinon l'état s' est considéré comme non visité et son tableau $history'$ prend alors la valeur $history \vee history'$.
2. dans le cas où $ample(s) = enabled(s)$, chaque successeur s' de s est placé dans une *waiting list* avec un tableau $history'$ réinitialisé avec toutes ses valeurs à faux. Dans le cas où s' n'appartient pas au même nœud que s, le tuple est alors envoyé à son propriétaire.

Le principe de cette seconde heuristique est de raffiner le mécanisme de détection de cycles "globaux" en marquant, pour chaque état, pour où sont passées les DFS qui l'ont exploré. Tant qu'une DFS ne tombe pas sur un état visité par une DFS étant passé par les même nœuds, l'état n'est pas considéré comme dangereux *pour cette DFS*. On considère alors qu'un état s peut fermer un cycle constitué d'états partiellement étendus uniquement lorsqu'une DFS atteignant s est passée par tous les nœuds par lesquels sont passées les autres DFS ayant exploré s.

Cette nouvelle heuristique permet d'améliorer l'efficacité des réductions mais elle nécessite aussi, dans certains cas, d'explorer plusieurs fois une même séquence d'états. Dans le pire cas, un état peut alors être visité N fois, où N représente le nombre de nœuds utilisés. Notons également, que le stockage du tableau pour chaque état introduit un léger surcoût mémoire de N bits par état.

6.3.3 Adaptation du proviso coloré

Nous allons présenter ici deux versions du proviso coloré en environnement réparti. La première version est une solution basique tirée du *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{ds}}$. La deuxième, légèrement différente, utilise véritablement le fait que les contraintes sur la pile soient relâchées.

a. Une première solution

Cette première solution est une solution basique qui interdit à un état non étendu de franchir une transition aboutissant à un changement de partition. Dans ce cas là, le nouveau *proviso*, noté $\mathrm{C2}^{\mathrm{L}}_{\mathrm{dc}\star}$, oblige un état s à être totalement étendu si une transition atteint un état rouge ou un état dans la pile pour lequel l'attribut *expanded* n'est pas inférieur à celui de s mais également lorsqu'un des successeurs de s n'appartient pas à la même partition que s.

Ce *proviso* aboutit à une restriction du *proviso* coloré aux seules explorations locales puisque que l'on utilise une approche particulièrement défensive pour ce qui est de la gestion des éventuels cycles répartis sur plusieurs partitions.

b. Une seconde solution : une meilleure utilisation des couleurs

L'intérêt de notre *proviso* coloré étant de relâcher les contraintes sur la pile et d'utiliser l'espace d'état, il semble alors plus normal d'essayer de ne pas tenter de résoudre le problèmes des cycles répartis uniquement *via* la pile en forçant l'état à être totalement étendu.

Dans cette seconde adaptation du *proviso* coloré, notée $\mathrm{C2}^{\mathrm{L}}_{\mathrm{dc}\star'}$, nous utilisons alors uniquement les couleurs d'états pour gérer les cycles répartis. Nous pouvons alors permettre à un état partiellement étendu d'avoir des successeurs appartenant à d'autres partitions.

```
Main ()
1   if id = manager then
2     waiting.push(s_0, empty_history);
3   end if
4
5   while not termination do
6     process_messages();
7     while waiting ≠ ∅ do
8       (state, history) ← pop(waiting);
9       if not visited(state, history) then
10        DFS(state, history);
11      end if
12      process_messages();
13    end while
14  end while

DFS (state, history)
1   if ample(s) = enabled(s) then
2     for t ∈ ample(s) do
3       if owner(t(s)) = id then
4         if not visited(t(state), history) then
5           push(waiting, t(state), empty_history);
6         end if
7       else
8         send(owner(t(s)), t(state), empty_history);
9       end if
10    end for
11  else
12    for t ∈ ample(s) do
13      if owner(t(s)) = id then
14        if not visited(t(state), history) then
15          DFS(t(state), history);
16        end if
17      else
18        send(owner(t(s)), t(state), history');
19      end if
20    end for
21  end if
```

FIG. 6.14 – Algorithme de Spin utilisant le *proviso* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{ds}}$.

Dans ce cas, l'état est coloré en rouge sauf, bien sûr, s'il est totalement étendu. Sa couleur est alors propagée à ses prédécesseurs comme dans l'algorithme séquentiel. Considérons alors les deux cas présentés à la figure 6.15. Dans le premier cas, les états $s_{0\ldots3}$ sont tous partiellement étendus. Comme s_2 possède un successeur appartenant à une autre partition, il est coloré en rouge. Sa couleur est propagée à ses prédécesseurs. Lorsque l'ensemble persistent pour s_3 est calculé, de la même façon, puisque son successeur appartient à une partition différente, il est coloré en rouge. Lorsque s_0 est revisité, étant coloré en rouge, s_3 devrait théoriquement être totalement étendu comme dans le cas de l'algorithme séquentiel. Pour éviter un surplus de communications, nous avons décidé de changer ce genre de situation assez particulière. Dans ce cas là, c'est l'état s_0 qui sera totalement étendu. On autorise alors ici un état stocké dans l'espace d'état à changer de couleur et passer du rouge au vert.
Trois cas de figure peuvent alors se présenter. Ils sont illustrés à la figure 6.15. Nous venons de présenter le premier cas. Le second cas illustré sur la figure 6.15 est, quant à lui, beaucoup plus simple, il décrit un comportement qui ne peut être obtenu en utilisant les *provisos* $\mathrm{C2}^{\mathrm{L}}_{\mathrm{ds}}$ ou $\mathrm{C2}^{\mathrm{L}}_{\mathrm{dc}\star}$: avec ces deux *provisos*, les états s_2 et s_3 auraient du être totalement étendus. Ici, en se basant

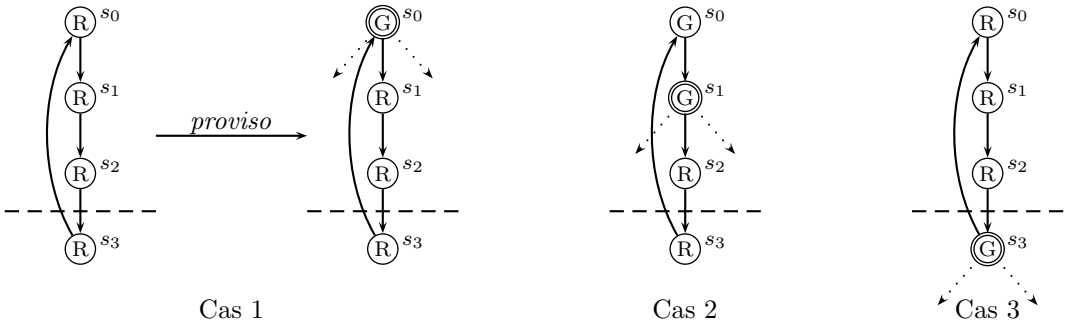


FIG. 6.15 – Algorithme basé sur le *proviso* $C2^{L}_{dc\star}$,

uniquement sur les couleurs et plus sur la pile, on permet de limiter le nombre d'états totalement étendus.
Le troisième cas en un peu particulier même s'il ressemble au premier cas. Ici, lorsque l'état s_3 est exploré, il est totalement étendu. Or un de ses successeurs s_0 appartient à une autre partition et est coloré en rouge. Si l'on se replace dans le cas 1, l'état s_0 devrait alors être totalement étendu inutilement ici. Pour éviter cela, lorsqu'un état totalement étendu possède des successeurs distants, un *tag* est adjoint au message pour signifier que si ce successeur est déjà visité, il n'est pas nécessaire de l'étendre totalement puisqu'il est le successeur d'un état sûr. On ne risque alors pas de refermer un cycle d'états partiellement étendus. Notons que ce cas là est déjà traité dans le cas séquentiel.

6.3.4 Evaluations

Les résultats présentés dans les tableaux 6.2, 6.3 et 6.4 présentent les résultats obtenus pour les différents *proviso* implémentés dans Cyclades.

Considérons alors le premier modèle utilisé : le système d'équilibrage de charge (*load balancer*). On observe ici qu'en séquentiel, le *proviso* $C2^{L}_{c\star}$ se comporte beaucoup mieux que $C2^{L}_{s}$. Lorsque l'on utilise le *proviso* $C2^{L}_{dc\star}$, basé uniquement sur la pile, on observe une dégradation notable de l'efficacité. Lorsque l'on passe à une approche moins agressive, on observe que le *proviso* $C2^{L}_{dc\star}$, est alors quasiment aussi efficace qu'en séquentiel. La raison de cette efficacité se trouve avant tout dans le degré de localité obtenu suite à un partitionnement structurel. En isolant une partie des cycles du réseau de Petri, on a alors pu isoler les cycles potentiels de l'espace d'état et les traiter de la même façon qu'en séquentiel. Les éventuels cycles répartis sur les sites distants sont alors la source d'un léger surplus d'états totalement étendus.
Observons ensuite le comportement des *provisos* $C2^{L}_{ds}$ et $C2^{L}_{ds'}$. Ici, la première surprise est que les réductions obtenues sont meilleures qu'en séquentiel ! Ce phénomène s'explique assez simplement. En modifiant le parcours global et les conditions sous lesquelles un état est totalement étendu, on a aboutit à des choix d'ensembles persistants différents de ceux obtenus en séquentiel. Dans certains cas, étendre un état plutôt qu'un autre peut grandement modifier la taille du graphe d'état réduit. Il est ainsi possible, dans certains cas, d'obtenir de meilleures réductions avec un *proviso* pour les propriétés de vivacité qu'avec un *proviso* pour les propriétés de sûreté pourtant beaucoup plus souple ! Ce phénomène particulier est observable dans notre cas où, couplé avec une fonction de partition permettant un fort degré de localité, le nombre des états totalement étendus n'est pas extrêmement élevé. Notons toutefois que notre *proviso* reste plus efficace en réparti ce qui confirme alors que pour ce modèle, le *proviso* $C2^{L}_{c\star}$ est bel et bien plus efficace que $C2^{L}_{s}$.

Considérons ensuite le second modèle analysé. Pour ce modèle, nous avons choisi d'utiliser un

partitionnement uniforme pour observer le comportement des différents *provisos* avec un degré de localité nul. Les résultats obtenus pour les *provisos* $C2^{L}_{ds}$ et $C2^{L}_{dc\star}$ sont alors tout à fait logique. Chaque état générant des successeurs appartenant à des sites distants, la totalité des états sont alors totalement étendus. Aucune réduction n'est alors observable. Le phénomène est alors similaire pour notre *proviso* $C2^{L}_{dc\star'}$ qui rend tous les états visités dangereux et aboutit alors, à chaque exploration d'état déjà visité, une exploration totalement étendue. Dans le contexte d'un degré de localité très faible, le comportement du *proviso* $C2^{L}_{ds'}$ est alors meilleur. Les résultats obtenus pour ce modèle sont alors exceptionnels.

Pour le modèle suivant (*Peterson*), les réductions obtenues avec les *provisos* $C2^{L}_{c\star}$ et $C2^{L}_{s}$ sont assez proches. On retrouve alors ce comportement pour les *provisos* $C2^{L}_{dc\star'}$ et $C2^{L}_{ds'}$. La principale différence en terme d'efficacité se trouve alors dans le nombre d'arcs considérés lors de la construction du graphe réduit. Ici, le nombre d'arcs explorés avec le *proviso* $C2^{L}_{ds'}$ est deux à trois fois supérieur à celui obtenus avec $C2^{L}_{dc\star'}$. Il est alors évident ici, que pour effectuer les réductions, $C2^{L}_{ds'}$ a été obligé d'effectuer des explorations redondantes.

Le modèle de l'allocateur de ressource est assez intéressant car en séquentiel, notre *proviso* aboutit à des réductions beaucoup plus efficaces que $C2^{L}_{s}$. Cette efficacité est ici conservée par $C2^{L}_{dc\star'}$ qui aboutit à des réductions nettement meilleures que $C2^{L}_{ds'}$. Là encore, l'explication réside dans le fort degré de localité obtenu en sélectionnant un cycle qui a permis d'utiliser au mieux $C2^{L}_{c\star}$ en local pour conserver toute son efficacité.

Considérons ensuite le modèle de l'algorithme d'élection. Ici, les réductions en séquentiel sont strictement identiques. La parallélisation aboutit alors à des réductions assez proches également et légèrement plus efficaces avec $C2^{L}_{ds'}$. Cependant, là encore, le nombre d'arcs visités est assez intéressant. Puisque l'on arrive pour certaines valeurs de n à des augmentations de plus de 1000% du nombre d'arcs explorés !
Nous pouvons donc observer que dès qu'un degré de localité convenable est atteint, notre *proviso* conserve des très bonnes performances et conserve sa meilleure efficacité comparé aux autres *provisos*. A performances égales (vis-à-vis des réductions obtenues), notre *proviso* génère beaucoup moins d'exploration d'arcs que le *proviso* $C2^{L}_{ds'}$ puisqu'il n'utilise pas de mécanisme d'exploration redondante.
Notons toutefois que lorsque le degré de localité est très faible, le *proviso* $C2^{L}_{ds'}$ reste la seule approche offrant des résultats acceptables comme le montrent les résultats obtenus pour le modèle du *Database Manager*.

6.4 Réductions d'ordre partiel en parallèle

Comme nous l'avons précisé au début du chapitre 4, la principale similitude entre le model checking dans un environnement à mémoire partagée et le model checking dans un environnement à mémoire répartie concerne la cohérence globale du parcours de l'espace d'état ou plutôt sa non-cohérence. Quelque soit le système considéré, le parcours de l'espace d'état n'a plus de cohérence globale dès que l'on parallélise l'exploration.

Dès lors, la problématique des réductions d'ordre partiel pour une recherche parallèle dans un environnement à mémoire partagée est identique à celle dans un environnement à mémoire répartie. Les solutions que l'on peut apporter sont alors exactement les mêmes.

Il est également envisageable d'adapter encore beaucoup plus finement notre *proviso* $C2^{L}_{c\star}$ en évitant de colorer des états en rouges alors qu'ils pourraient ne pas l'être. Pour cela, les conditions sous lesquelles l'exploration d'un état est déléguée à une autre *thread* pourraient être assouplies en privilégiant la délégation des états verts. On limiterait alors les états "faussement dangereux", i.e. les états colorés en rouges alors qu'ils sont normalement sûrs.

Load Balancer *(Structural Partitionning)*

No POR : *9 613 008 states* *51 203 400 arcs*
$C2^L_s$ *:* *3 139 224 states* *6 986 331 arcs*
$C2^L_{c\star}$ *:* *525 629 states* *913 451 arcs*
PS : *524 414 states* *909 896 arcs*

	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	3 140 220	6 663 348	82	1 882 987	3 854 473	81	1 329 441	3 240 847	81	528 479	928 466	71
3	664 267	1 456 190	57	661 923	1 444 534	57	531 134	982 328	66	540 270	985 567	49
4	2 956 660	7 064 285	55	905 617	1 842 819	44	1 907 241	4 757 271	60	529 927	935 209	59
5	3 170 245	7 966 750	35	716 025	1 712 392	30	1 907 241	4 757 271	47	541 456	987 903	37
6	1 848 859	4 498 576	42	995 604	2 093 598	42	1 336 161	3 258 417	56	539 959	995 421	38
7	3 315 783	8 748 770	32	753 225	1 920 312	27	1 907 241	4 757 271	46	542 551	990 446	36
8	3 183 668	8 179 570	39	694 489	1 608 467	19	1 907 241	4 762 243	50	535 110	962 701	44
9	2 081 271	5 491 570	27	763 730	2 011 568	23	1 332 927	3 428 174	48	541 986	997 134	33
10	3 303 262	8 759 550	32	728 382	1 792 676	15	1 907 241	4 954 154	46	542 314	992 761	34

Database Manager *(Uniform Partitionning)*

No POR : *6 908 734 states* *55 269 890 arcs*
$C2^L_s$ *:* *372 737 states* *639 002 arcs*
$C2^L_{c\star}$ *:* *372 737 states* *639 002 arcs*
PS : *372 737 states* *639 002 arcs*

	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	6 908 734	55 269 890	0	375 297	646 864	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
3	6 908 734	55 269 890	50	376 148	657 005	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
4	6 908 734	55 269 890	0	373 754	642 786	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
5	6 908 734	55 269 890	0	374 576	646 777	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
6	6 908 734	55 269 890	0	373 315	641 425	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
7	6 908 734	55 269 890	0	373 782	643 717	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
8	6 908 734	55 269 890	0	373 173	640 899	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
9	6 908 734	55 269 890	0	373 416	642 260	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0
10	6 908 734	55 269 890	0	373 074	640 595	0	6 908 734	55 269 890	0	6 908 734	55 269 890	0

TAB. 6.2 – Comparaison des *provisos* implémentés dans Cyclades (1)

Peterson *(Structural Partitionning)*

No POR : *1 242 528 states* *4 970 112 arcs*
$C2^L_s$ *:* *240 059 states* *583 604 arcs*
$C2^L_{c\star}$ *:* *186 968 states* *387 506 arcs*
PS : *186 968 states* *387 506 arcs*

	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	436 161	941 757	67	215 043	646 324	68	410 797	862 555	66	187 295	388 652	70
3	533 099	1 161 212	59	213 385	746 894	59	515 008	1 100 629	59	188 115	390 640	62
4	549 019	1 200 391	53	209 383	833 711	51	542 462	1 178 258	53	188 408	391 391	56
5	925 079	2 880 380	32	230 540	954 057	38	924 605	2 875 130	32	262 074	730 114	32
6	607 484	1 321 052	49	206 120	921 691	44	607 476	1 321 035	49	188 465	391 552	50
7	1 022 038	3 362 152	26	225 822	1 029 166	34	1 022 038	3 362 143	26	271 302	778 198	29
8	921 499	2 515 117	34	211 294	985 320	37	921 496	2 515 105	34	204 910	459 401	37
9	985 726	2 828 134	32	207 697	993 332	36	985 721	2 828 120	32	221 618	536 677	34
10	1 071 060	3 308 220	28	208 344	1 008 105	34	1 071 060	3 308 211	28	266 739	756 063	29

Allocator *(Structural Partitionning)*

No POR : *3 628 728 states* *24 807 270 arcs*
$C2^L_s$ *:* *2 905 796 states* *10 439 108 arcs*
$C2^L_{c\star}$ *:* *413 936 states* *874 880 arcs*
PS : *22 362 states* *36 468 arcs*

	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	1 100 101	2 916 255	83	1 151 457	2 969 322	78	778 348	1 813 808	85	413 936	874 880	82
3	1 148 262	3 177 418	74	1 643 112	3 071 941	75	1 040 689	2 628 341	76	424 546	884 264	75
4	1 822 341	5 499 972	65	1 675 102	4 728 449	57	1 612 931	4 477 277	68	414 218	882 214	69
5	2 427 210	8 342 642	64	1 737 996	5 221 223	60	2 222 789	6 870 385	66	425 236	882 128	66
6	1 658 181	4 837 521	62	1 143 921	3 645 669	56	1 658 181	4 837 521	62	442 711	896 452	63
7	2 797 286	11 010 710	44	1 937 890	5 860 811	42	2 797 286	11 010 710	44	456 784	936 452	45
8	2 374 691	8 092 407	51	1 737 100	5 456 531	47	2 374 691	8 092 407	51	468 157	942 153	50
9	2 487 291	8 803 727	50	1 795 190	5 937 340	46	2 487 291	8 803 727	50	491 574	964 741	50
10	2 457 248	8 524 932	53	1 701 268	5 765 258	51	2 457 248	8 524 932	53	512 216	997 719	53

TAB. 6.3 – Comparaison des *provisos* implémentés dans Cyclades (2)

	Leader election Protocol											*(Structural Partitionning)*
						No POR :		*10 475 430 states*				*117 722 093 arcs*
						$C2_s^L$ *:*		*7 278 309 states*				*48 636 169 arcs*
						$C2_{c\star}^L$ *:*		*7 278 309 states*				*48 636 169 arcs*
						PS :		*7 278 309 states*				*48 636 169 arcs*
	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	7 245 571	73 047 696	65	7 954 624	197 421 246	65	7 245 571	73 047 696	65	7 651 992	110 285 585	59
3	9 025 622	97 832 973	20	7 717 346	212 142 011	20	9 025 622	97 832 973	20	8 288 138	120 981 130	17
4	8 354 534	88 667 713	32	7 477 699	220 043 055	30	8 354 534	88 667 713	32	8 022 374	121 620 415	29
5	9 753 382	107 904 061	15	7 442 621	254 296 783	13	9 753 382	107 904 061	15	8 352 833	130 995 666	13
6	9 025 622	98 041 869	15	7 738 441	284 559 099	13	9 025 622	98 041 869	15	8 295 453	126 749 351	12
7	10 107 558	112 671 533	12	7 777 028	377 492 318	12	10 107 558	112 671 533	12	8 414 854	128 801 305	11
8	9 461 638	104 083 517	14	7 599 183	487 168 736	14	9 461 638	104 083 517	14	8 396 518	127 783 232	13
9	10 285 990	115 110 893	5	7 627 349	532 246 641	5	10 285 990	115 110 893	5	8 428 424	130 412 420	4
10	9 753 382	108 096 573	12	7 328 147	557 543 284	12	9 753 382	108 096 573	12	8 429 808	130 496 056	10

	Production cell											*(Structural Partitionning)*
						No POR :		*states*				*arcs*
						$C2_s^L$ *:*		*923 166 states*				*1 278 723 arcs*
						$C2_{c\star}^L$ *:*		*185 647 states*				*221 486 arcs*
						PS :		*184 026 states*				*218 346 arcs*
	$C2^L_{ds}$			$C2^L_{ds'}$			$C2^L_{dc\star}$			$C2^L_{dc\star'}$		
n	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d	$\|S\|$	$\|A\|$	d
2	424 205	523 475	95	414 147	527 997	94	351 271	419 044	97	339 719	403 820	97
3	365 489	438 790	96	352 787	441 023	95	354 294	422 884	97	339 854	403 854	96
4	359 249	429 463	96	343 877	430 799	95	357 023	426 266	96	339 695	403 430	95
5	494 165	603 955	93	343 877	437 990	91	492 083	600 902	93	339 695	403 430	91
6	359 249	429 463	97	343 877	430 627	96	357 023	965 918	97	339 695	403 430	96
7	494 165	603 956	94	343 877	437 947	91	492 083	600 902	94	339 695	403 430	91
8	450 188	546 756	94	342 174	435 970	93	450 133	546 678	94	339 877	403 612	93
9	472 694	575 858	94	342 174	437 278	91	472 643	575 784	94	339 877	403 612	91
10	495 200	604 960	94	342 174	438 477	90	495 153	604 890	94	339 877	403 612	90

TAB. 6.4 – Comparaison des *provisos* implémentés dans Cyclades (3)

Les algorithmes présentés n'ont toutefois pas encore été adaptés pour la version de parallèle Cyclades pour les systèmes multiprocesseurs.

6.5 Réductions d'ordre partiel en réparti multithreadé

Dans un contexte parallèle et réparti, les seules solutions utilisables sont celles du contexte réparti. Là encore nous n'avons pas adapté nos algorithmes à cet environnement. Cependant, on peut espérer, pour le cas du *proviso* de Spin $C2^{L}_{ds'}$, de bons résultats en terme de gain de temps du fait des nombreuses explorations redondantes. On aboutirait alors à des séquences d'exploration plus importantes et donc une meilleure efficacité.

6.6 Conclusion

Dans ce chapitre, nous avons commencé par présenter rapidement le fonctionnement des méthodes d'ordre partiel. Puis nous avons présenté nos travaux concernant deux nouveaux algorithmes alternatifs pour l'implémentation du problème d'ignorance dans un environnement séquentiel. Ces algorithmes permettent d'augmenter l'efficacité des réductions sur de nombreux modèles.

Nous nous sommes ensuite attaché à présenter la difficulté d'adapter ces méthodes à un contexte parallèle. Cette difficulté découle du principe même des algorithmes de réductions d'ordre partiels qui sont tous basés sur la cohérence du parcours de l'espace d'état. C'est cette nécessité de cohérence qui limite fortement l'adaptation de ces méthodes dès que l'on veut paralléliser l'exploration puisque la cohérence globale du parcours ne peut plus être assurée.

Deux approches sont possibles, qui ne sont pas sans rappeler les deux approches proposées pour l'adaptation de la technique des Δ-*markings* dans un environnement répartie. La première méthode consiste à se baser sur notre algorithme utilisant la notion de *couleur d'état*. Ces couleurs permettent de relâcher les contraintes sur l'exploration et l'utilisation exclusive de la pile d'exploration et de s'appuyer un peu plus sur l'espace d'état lui-même. Cette méthode est une adaptation qui ne préserve cependant pas toujours l'efficacité des réductions.

La seconde approche est celle utilisée par Spin, elle consiste à essayer de s'approcher au plus de l'efficacité des réductions en environnement séquentiel en contrepartie d'une certaine efficacité en terme de temps d'exécution. Dans cette approche, l'idée consiste à retrouver un peu de cohérence globale dans le parcours en permettant notamment des explorations redondantes qui vont sérieusement dégrader les gain de temps potentiels. Ce surcoût pourrait alors être réduit par une version parallèle et répartie qu'il reste encore à implémenter.

Troisième partie

Outils de vérification et d'analyse de programmes concurrents

Chapitre

7

La plateforme Quasar

Sommaire

Quasar [113] est une plate-forme de vérification automatique de programmes Ada concurrents basé sur le formalisme des réseaux de Petri colorés. Cette plate-forme a été l'objet de deux thèses [31, 103] en plus de celle-ci. Quasar est le fruit de recherches et d'implémentations effectuées en collaboration avec Olivier Alzéari, Sami Evangelista, Claude Kaiser, Jean-François Pradat-Peyre, Pierre Rousseau et Nicolas Trèves.

Dans un premier temps, nous présenterons rapidement l'architecture de la plate-forme (7.1). Nous effectuerons ensuite une rapide présentation des principales fonctionnalités du langage Ada (7.2) avant d'aborder la méthode de traduction d'un programme Ada concurrent en un réseau de Petri coloré de haut-niveau (7.3). Nous nous attarderons ensuite sur l'un des travaux liminaires de cette thèse, à savoir : la modélisation de tâches créées dynamiquement (7.4).

7.1 La plate-forme Quasar

Quasar divise le processus de vérification de programme en 4 phases distinctes totalement automatique et effectuées par autant d'outils. L'utilisateur fournit simplement un programme Ada ainsi qu'une propriété.

1. **découpe** (ou *slicing*) de programme concurrent : cette première étape est réalisée par l'outil Yasnost, elle consiste à créer une *abstraction* P' d'un programme P vis-à-vis d'une propriété Q. Cette découpe permet de supprimer dans le programme P tous les éléments inutiles à la vérification de Q. Le programme réduit P' obtenu est donc équivalent vis-à-vis de la propriété Q au programme initial. Cette première étape permet donc de réduire le programme a analysé et constitue ainsi une première attaque sur le problème de l'explosion combinatoire.
   Plus de détails sur cette première phase et sur l'outil Yasnost peuvent être trouvés dans [104] et [103].
2. **traduction** et **réduction** : cette seconde étape consiste à traduire le programme réduit obtenu en un réseau de Petri coloré R. Cette traduction se base sur la notion de *patterns*. Nous reviendrons plus en détail sur cette étape dans les sections 7.3 et 7.4. Suite à cette traduction, un certain nombre de réductions structurelles sont effectuées sur le réseau [24, 18, 22] pour obtenir un réseau réduit R'.
3. **vérification** (ou model checking) : cette étape permet d'explorer le graphe d'accessibilité du réseau R' afin de vérifier la validité de la propriété Q. Cette étape est effectuée à l'aide de l'outil Helena ou Cyclades que nous présenterons plus en détails dans le chapitre suivant.
4. **rapport d'erreur** : cette dernière étape permet d'afficher un rapport d'erreur détaillé à l'utilisateur dans le cas où la propriété n'est pas vérifiée. A cette étape il est alors nécessaire d'effectuer le travail inverse de la seconde étape, à savoir : partir du rapport d'erreur sur le réseau de Petri pour le faire correspondre au programme P initial. Cette étape, particulièrement compliquée, est encore en cours de développement.

L'architecture de la plate-forme est présentée à la figure 7.1. Comme on peut le voir, à chaque étape correspond un outil totalement indépendant. La première étape de *slicing* génère un programme Ada réduit passé ensuite au traducteur. Ce dernier génère un réseau de Petri qui peut être analysé tant par Helena que par Cyclades. Ces deux outils peuvent alors générer un rapport d'erreur qui sera traité puis affiché par l'interface graphique.

7.2 Le langage Ada

Ada est un langage de programmation de haut niveau particulièrement adapté au développement d'applications critiques qui requièrent un haut niveau de fiabilité. Il est l'un des rares langages à avoir été normalisé par l'ISO, (International Standardisation Organism), un organisme de normalisation internationale.
Nous ferons dans cette section un bref rappel historique sur la naissance de la version utilisée du langage Ada, appelée *Ada 95*, et de son prédécesseur *Ada 83*. Puis nous rappellerons les caractéristiques principales du langage ainsi que les mécanismes de concurrence qu'il offre.

7.2.1 Historique

Le développement du langage Ada est historiquement lié au département de la défense américaine (DoD) qui souhaitait se doter d'un langage de programmation de haut niveau adapté au développement d'applications embarquées critiques. A ce titre, le DoD parraine les nombreuses phases de définition et de développement du langage : définition d'un cahier des charges, développements, évaluations, ... Ce processus débute en 1974 et s'achève en 1983 avec la publication du standard ANSI (American National Standards Institute) Ada 83.
Le nom Ada provient de Augusta Ada Byron, comtesse de Lovelace (1815-1852). Ada, la fille de Lord Byron, était l'assistante et le mécène de Charles Babbage et travaillait sur la machine

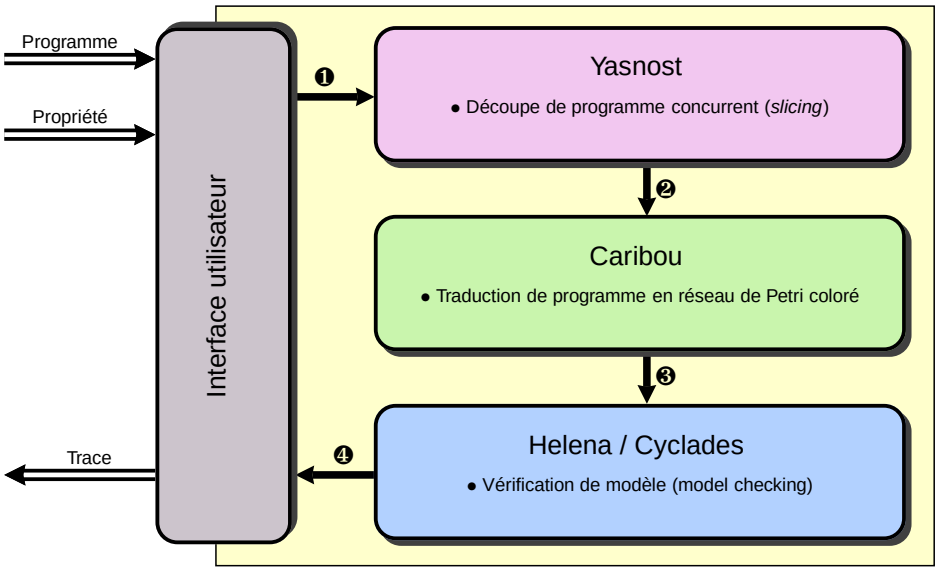


FIG. 7.1 – Architecture de la plate-forme Quasar.

analytique mécanique. Ada Lovelace peut donc être considérée à juste titre comme le premier programmeur de l'histoire.
En 1987, Ada 83 est finalement normalisé par l'ISO et devient la norme ISO 8652. Depuis la publication de cette norme, le groupe de travail WG9 de l'ISO se réunit régulièrement afin de discuter d'éventuelles modifications à apporter au langage. De ces réunions naît le 15 février 1995 la norme Ada 95 dont la définition officielle est le *Reference Manual for the Ada Programming Language*. Ada 95 est constitué d'un noyau et d'un certain nombre d'annexes spécialisées. Tout compilateur doit implémenter ce noyau afin d'être certifié ISO 8652, mais la norme n'impose pas l'implémentation des annexes. Depuis la publication d'Ada 95, le groupe WG9 a poursuivi son travail et a terminé une nouvelle révision du langage, Ada 2005, qui apporte quelques extensions au langage.

Remarque. Dans la suite de ce document, nous entendrons par Ada le langage Ada 95 défini dans [136].

7.2.2 Caractéristiques du langage

L'objectif principal d'Ada est le génie logiciel. On peut considérer que le génie logiciel pose deux problèmes : la nécessité de réutiliser, autant que possible, les composants logiciels et le besoin de définir des méthodes de travail.
En tant que langage, Ada fournit, dans une large mesure, la technologie permettant l'écriture de logiciels réutilisables grâce à la possibilité de définir des interfaces, et grâce au concept de généricité.
Le typage fort et les fonctionnalités s'y rapportant garantissent que les programmes contiendront peu de surprises : la plupart des erreurs seront détectées à la compilation. Nombre des erreurs restantes seront détectées par des contraintes additionnelles ajoutées par le compilateur. De plus, les vérifications à la compilation passent la frontière des unités de compilation.
Les fonctionnalités apportées par Ada 95 qui contribuent à une meilleure flexibilité sont les éléments de la programmation orienté objet, les types étendus et étiquetés, les bibliothèques hiérarchiques et une grande facilité concernant la manipulation des pointeurs et des références. Ceux-ci sont des outils puissants pour une programmation par extension.
En matière de concurrence Ada 95 a introduit les types protégés. On obtient ainsi une implémen-

tation plus naturelle et plus efficace des accès à des données partagées. En effet, dans la première version du langage, il était nécessaire d'introduire des tâches afin de contrôler ces accès.

7.2.3 La concurrence dans Ada

En matière de concurrence, le langage Ada fait preuve d'originalité puisque celle-ci est une partie intégrante du langage. Ada diffère donc des nombreux langages qui "sous-traitent" la concurrence au système d'exploitation. Intégrer la concurrence dans le langage a un avantage considérable. Cela permet en effet de rendre les applications concurrentes portables en évitant de placer dans les programmes les appels systèmes nécessaires à la mise en oeuvre de la concurrence. Tous ces appels seront exclusivement gérés par le compilateur.
L'objectif de cette section est de présenter brièvement la concurrence dans Ada. Dans un premier temps, nous décrirons les notions de tâches et d'objets protégés que nous regrouperons sous l'appellation *types concurrents*. Nous verrons ensuite les mécanismes fournis par le langage qui permettent la synchronisation entre tâches.
Nous ne traiterons pas tous les aspects du langage qui touchent à la concurrence. Pour obtenir de plus amples détails sur ces questions, le lecteur pourra consulter la section 9 du manuel de référence [136] ainsi que les ouvrages [132] et [133] qui sont entièrement dédiés à ce sujet.

a. Les types concurrents

i. Les tâches

En Ada, les processus se nomment des tâches. D'un point de vue syntaxique la déclaration d'une tâche ressemble fortement à la déclaration d'un paquetage : une tâche est constituée d'une spécification décrivant les points d'entrées de la tâche, i.e., l'interface présentée aux autres tâches, et d'un corps (ou **body**) qui décrit le comportement de la tâche, i.e., la séquence d'instructions exécutée par la tâche. Dans l'exemple suivant, nous déclarons une tâche Porte ayant deux points d'entrée Ouvre et Ferme.

```
task Porte is                task body Porte is
   entry Ouvre;              begin
   entry Ferme;                 ...
end Porte;                   end Porte;
```

Afin de pouvoir créer plusieurs tâches ayant le même comportement, Ada fournit la possibilité de déclarer un type tâche. Dans ce cas, la spécification du type tâche commence par les mots-clés **task type** (au lieu du simple mot clé **task**). Le corps du type tâche suit, quant à lui, la même syntaxe que le corps d'une tâche.

La création, i.e., l'élaboration, d'une tâche peut se faire de trois manières différentes.
– La spécification d'une tâche entraîne sa création.
– La déclaration d'une variable d'un type tâche T (ou d'un type composite incluant récursivement un type tâche T) entraîne la création d'une tâche de type T.
– Enfin, une tâche peut être activée par un allocateur. Si T est un type tâche, l'instruction suivante entraîne l'activation d'une tâche de type T :

```
Var := new T;
```

Dans les deux premiers cas nous dirons que la tâche est élaborée statiquement. Dans le troisième cas, on parle d'activation dynamique.

ii. Les objets protégés

Les objets protégés n'étaient pas présents dans la première version du langage et forment un des principaux ajouts à Ada en 1995. Les objets et types protégés sont basés sur l'encapsulation des données, accessibles uniquement via des sous-programmes ou des entrées protégées. La sémantique

du langage Ada 95 garantit que ces sous-programmes et ces entrées sont exécutées de façon à assurer la modification des données encapsulées en exclusion mutuelle. Les entrées protégées permettent de plus de réaliser des mises en attente de tâches sur des conditions appropriées, plus précisément sur des conditions de synchronisation. Comme pour les entrées de tâche (cf. plus loin), chaque entrée protégée possède une file d'attente qui contient l'ensemble des tâches bloquées sur un appel de l'entrée.
La déclaration d'un objet ou d'un type protégé est similaire à la déclaration d'une tâche. Un objet protégé possède une interface qui consiste en la liste des sous-programmes et entrées qui permettent d'accéder en lecture (pour les fonctions) ou en lecture / écriture (pour les procédures et entrées protégées) aux données encapsulées dans l'objet. Ces données doivent nécessairement apparaître dans la partie privée de la spécification, puisque seuls les sous-programmes et entrées de l'objet protégé peuvent manipuler ces données. Notons de plus qu'une fonction ne peut pas modifier les données encapsulées dans l'objet protégé. Le corps de l'objet contient l'implémentation des sous-programmes et entrées déclarées dans la spécification de l'objet.

Dans le programme donné ci-dessous suivant, nous déclarons un objet protégé Mutex possédant une procédure Rendre et une entrée protégée Prendre. L'exécution de l'entrée Prendre est gardée par la condition booléenne Libre. Par conséquent, l'entrée dans Prendre ne sera possible que si la variable Libre de l'objet protégé est positionnée à True.

```
protected Mutex is
   entry Prendre;
   procedure Rendre;
private
   Libre : Boolean := True;
end Mutex;
```

```
protected body Mutex is
   entry Prendre when Libre is
   begin
      Libre := False;
   end;
   procedure Rendre is
   begin
      Libre := True;
   end;
end Mutex;
```

b. Les mécanismes de synchronisation

i. Les rendez-vous

La spécification d'une tâche (ou d'un type tâche) décrit l'interface que la tâche présente aux autres tâches. Celle-ci consiste en une liste d'entrées qui sont des services que la tâche peut rendre aux autres tâches du programme.
Toute tâche peut *appeler* une entrée, i.e., demander un service, d'une autre tâche si celle-ci apparaît dans la partie publique de sa spécification. La tâche appelante est alors en attente de l'acceptation par le serveur de l'entrée appelée. La tâche appelée peut choisir *d'accepter* l'appel. On utilise pour cela l'instruction composite **accept**. Celle-ci exécute alors le corps de l'instruction. Une fois cette exécution terminée, elle acquitte la tâche appelante, et toutes deux peuvent poursuivre leurs exécutions.
Ce mécanisme de synchronisation porte le nom de rendez-vous entre tâches ou plus simplement *rendez-vous*. Dans le programme suivant nous présentons un exemple de rendez-vous entre une tâche Client et une tâche Serveur.

```
task Serveur is
   entry Service;
end Serveur;
task body Serveur is
begin
   loop
      accept Service;
```

```
task Client;
task body Client is
begin
   loop
      Serveur.Service;
   end loop;
end Serveur;
```

```
   end loop;
end Serveur;
```

Il peut arriver qu'une tâche accepte une entrée pour laquelle aucun appel n'a été fait. Dans ce cas la tâche est bloquée jusqu'à ce qu'une autre tâche appelle l'entrée correspondante. Une fois cet appel effectué, le rendez-vous peut avoir lieu.
Les appels pendants sur une entrée sont placés dans une file d'attente. Notons que la norme ne définit pas quelle tâche doit être choisie lorsqu'une tâche appelée a la possibilité de traiter plusieurs requêtes, i.e., lorsque la taille de la file d'attente est strictement supérieure à 1. Par conséquent, le choix de la demande à satisfaire peut porter sur toute demande de rendez-vous.

Ada donne la possibilité d'écrire des formes de rendez-vous plus évoluées. En particulier une tâche peut vouloir accepter des demandes de rendez-vous de manière inderministe. Pour cela l'instruction **select** permet de mettre une tâche en attente sur plusieurs entrées. Lorsqu'une demande de rendez-vous arrive sur une des entrées, la tâche peut immédiatement accepter la demande et procéder au traitement de l'instruction **accept** correspondante.
Le programme ci-dessous contient un exemple d'instruction **select**. Notons que les deux branches du **select** sont *gardées* par des conditions booléennes. A l'exécution du **select**, toutes ces gardes sont évaluées, et seules les branches pour lesquelles la garde est vérifiée seront considérées.

```
task Buffer is
   entry Lire (Valeur : out Integer);
   entry Ecrire (Valeur : in Integer);
end Buffer;

task body Buffer is
   Val   : Integer;
   Plein : Boolean := False;
begin
   loop
      select
         when Plein =>
            accept Lire (Valeur : out Integer) do
               Plein := False; Valeur := Val;
            end Lire;
      or
         when not Plein =>
            accept Ecrire (Valeur : in Integer) do
               Plein := True; Val := Valeur;
            end Ecrire;
      end select;
   end loop;
end Buffer;
```

Il existe trois autres formes d'instruction **select** que nous présentons brièvement : l'appel d'entrée temporisé, l'appel d'entrée immédiat et le transfert de contrôle asynchrone. L'appel d'entrée temporisé permet à une tâche appelante d'annuler une demande de rendez-vous si l'appel n'a pas été accepté après un certain délai. Dans le même esprit, l'appel d'entrée immédiat permet à une tâche appelante d'annuler sa demande de rendez-vous si la tâche appelée n'est pas bloquée sur une instruction **accept** sur l'entrée correspondante. Enfin le transfert de contrôle asynchrone permet d'avorter un calcul si une condition particulière est remplie (l'acceptation d'un rendez-vous ou l'écoulement d'un délai).

ii. Les opérations protégées

Les sous-programmes et entrées qui apparaissent dans la spécification d'un objet protégé ou d'un type protégé peuvent être appelés par les tâches du programme. On parle alors d'*opération protégée*. Un tel appel se fait en précisant le nom de l'objet protégé, le nom de la fonction, procédure, ou

entrée, ainsi que la liste des paramètres. Concernant la sémantique d'un tel appel, on considère trois cas.

- Si le service appelé est une fonction, l'entrée dans la fonction correspondante a lieu si aucune autre tâche n'exécute actuellement une procédure ou une entrée du même objet. Par conséquent, l'exécution d'une fonction sur l'objet protégé n'exclut pas l'exécution par une autre tâche d'une fonction sur ce même objet. Les fonctions protégées n'ayant pas d'effet de bord sur les paramètres de l'objet, la modification des données encapsulées en exclusion mutuelle est ainsi vérifiée.
- Si le service appelé est une procédure, l'entrée dans celle-ci a lieu si aucune autre tâche n'exécute actuellement un sous-programme (procédure ou fonction) ou une entrée de l'objet.
- L'appel d'une entrée protégée provoque l'exécution de celle-ci si les deux conditions suivantes sont remplies :
  1. Aucune tâche n'exécute actuellement un sous-programme ou une entrée de l'objet.
  2. La garde de l'entrée protégée est évaluée à True.

  Si une de ces deux conditions n'est pas remplie, la tâche appelante est placée dans la file d'attente de l'entrée protégée.

Lorsqu'une opération protégée est terminée, une tâche bloquée sur un appel portant sur le même objet est choisie de manière arbitraire parmi celles pour lesquelles la garde est devenue vraie. Si les conditions précédemment évoquées sont remplies, la tâche peut s'exécuter et une nouvelle opération protégée débute sur l'objet.

Ada distingue les appels externes des appels internes. Un appel externe est exécuté hors du corps l'objet. Les conditions que nous avons énoncées concerne les appels externes. Dans le cas d'un appel interne, i.e., réalisé dans un des sous-programmes ou entrées de l'objet, sur l'objet lui même, la sémantique est différente. On considère alors que l'appel peut être traité sans condition. Cependant, dans la cas des entrées protégées, aucune distinction n'est faite entre un appel externe et un appel interne. Par conséquent, un appel interne sur une entrée protégée est toujours bloquant, puisque la tâche tentera de reprendre un verrou en écriture qu'elle possède déjà.

iii. La remise en queue

Conjointement au concept d'objet protégé, la norme Ada 95 a introduit l'instruction de remise en queue **requeue**. L'objectif était de pallier une des limitations majeures des entrées de tâches et entrées protégées : les conditions de synchronisation additionnelles placées dans les gardes d'entrées protégées ou de **select** ne peuvent pas porter sur les paramètres de l'entrée.
Une instruction **requeue** peut être placée dans le corps d'une entrée protégée ou dans le corps d'une instruction **accept**, d'acceptation d'entrée de tâche. Le mécanisme concrétisé par l'instruction **requeue** peut être utilisé pour compléter un corps d'entrée, en redirigeant par la suite l'appel correspondant vers une nouvelle entrée (protégée ou de tâche). En d'autres termes, nous allons pouvoir, par le biais de ce mécanisme, compléter les conditions d'acceptation et mettre à jour les variables d'état avant de replacer la tâche appelante en attente passive sur la file d'attente associée à l'entrée désignée par l'instruction **requeue**.
L'instruction **requeue** est généralement utilisée de la manière suivante :

```
entry E (P1 : Param_Type1; ...; PN : Param_TypeN) when Condition is
begin
   if Condition2 then
      Traitement;
   else
      requeue E2;
   end if;
end;
```

La première condition d'entrée Condition ne pouvant pas porter sur les paramètres de l'entrée E, les tests nécessaires portant sur ces paramètres sont effectués dans le corps de l'entrée (condition Condition2). Si cette condition supplémentaire n'est pas vérifiée, la requête est redirigée vers l'entrée E2.

iv. Les familles d'entrée

Les familles d'entrée sont une solution simple et élégante pour représenter un ensemble d'entrées. Illustrons cette construction à l'aide d'un exemple. Supposons un objet protégé acceptant des requêtes de priorités diverses, identifiées par le type suivant :

```
type Priorite is (Faible, Normal, Urgent);
```

La solution suivante résout le problème en définissant trois entrées, une pour chaque priorité. Notons l'utilisation de l'attribut Count qui compte le nombre de tâches présentes dans la file d'attente d'une entrée protégée (ou de tâche). Nous exprimons donc le fait que l'exécution d'une requête est conditionnée par l'absence de requêtes de priorité plus élevée.

```
protected Controleur is
  entry Req_Faible;
  entry Req_Normale;
  entry Req_Urgente;
end Controleur;
```

```
protected body Controleur is
  entry Req_Faible  when Req_Normale'Count=0
                     and Req_Urgente'Count=0 is...
  entry Req_Normale when Req_Urgente'Count=0 is...
  entry Req_Urgente when True is...
end Controleur;
```

Cette solution est lourde et inélégante. De plus, elle dépend directement de la définition du type Priorite : si nous modifions ce type, il faudra modifier l'objet protégé en conséquence.
La définition d'une famille d'entrées permet de répondre à notre besoin. Nous allons pour cela définir la famille Req indicée par le type Priorite. La fonction Plus_Prioritaire permet de tester si une requête de priorité supérieure au paramètre P est effectuée. Elle est utilisée dans la garde de la famille Req. Tous l'intérêt est de pouvoir utiliser l'indice P de la famille dans l'expression de garde. Cette solution est donc plus légère et ne dépend pas de la définition du type Priorite.

```
protected Controleur is
   entry Req (Priorite);
end Controleur;
protected body Controleur is
   function Plus_Prioritaire (P : in Priorite) return Boolean is
   begin
      if P = Priorite'Last then return False; end if;
      for Q in Priorite range Priorite'Succ (P)..Priorite'Last loop
         if Req (Q)'Count > 0 then return True; end if;
      end loop;
      return False;
   end;
   entry Req (for P in Priorite) when not Plus_Prioritaire (P) is ...
end Controleur;
```

7.3 Traduction de programmes Ada en réseaux de Petri colorés

Les réseaux de Petri (avec arcs inhibiteurs) ayant le même pouvoir d'expression qu'une machine de Turing il n'existe aucun obstacle théorique à la traduction d'un programme en un réseau coloré. De plus, avec les restrictions que nous avons présentées dans la section précédente il est possible de se passer des arcs inhibiteurs. Tout programme Ada ayant le profil Quasar peut donc être traduit en un réseau coloré tel que défini dans le chapitre 1.

La traduction d'un programme concurrent en un réseau de Petri coloré est similaire à la compilation d'un programme en langage assembleur. Dans les deux cas, le but est de traduire la sémantique du programme d'un formalisme vers un autre.
Quasar procède de la manière suivante pour générer un réseau de Petri à partir d'un programme Ada. Nous associons à chaque construction du langage, (p.ex., déclaration de variable, entrée

protégée) un sous-réseau ou patron. Durant l'analyse syntaxique du programme, Quasar génère le sous-réseau correspondant à chaque élément du programme. Une fois la traversée de l'arbre syntaxique terminée, nous obtenons un ensemble de sous-réseaux qui sont combinés ensemble pour produire le réseau coloré final. Cette approche a le mérite de faciliter la preuve de correction du processus de traduction. Il suffit en effet de vérifier que chaque patron respecte bien la sémantique du langage. Les réseaux hiérarchiques nous ont paru adapté pour définir ces patrons.

7.3.1 Les réseaux hiérarchiques

Les réseaux hiérarchiques ont été introduits par Huber, Jensen et Shapiro dans [6]. L'objectif, lors de la conception de ces réseaux était d'appliquer les méthodes de génie logiciel au processus de modélisation de systèmes répartis à l'aide des réseaux de Petri colorés. L'introduction de hiérarchies permet de concevoir un réseau comme un ensemble de modules ou pages réutilisables et qui peuvent être composés à l'aide de divers opérateurs.
Nous présentons maintenant de manière informelle les réseaux hiérarchiques manipulés par Quasar et qui assez similaires à ceux de [6]. Les différents éléments de nos réseaux hiérarchiques sont les réseaux composites et les méta-transitions. Ceux-ci peuvent être composés par substitution de transitions et par fusion de places ou réseaux pour obtenir un réseau de Petri coloré standard.

Un **réseau composite** est un réseau coloré qui peut lui-même être décomposé en réseaux composites et dans lequel les transitions peuvent être des transitions de substitution. Une telle transition est une abstraction dans la mesure où elle n'apparaîtra pas dans le réseau coloré final. Au niveau du réseau composite, l'effet de cette transition n'est pas détaillé et est caché au concepteur du modèle.
La **méta-transition** est un réseau composite dans lequel nous distinguons deux transitions : la transition d'entrée et la transition de sortie. Les méta-transitions sont utilisées pour détailler les transitions de substitution d'un réseau composite de niveau supérieur. Les transitions d'entrée et de sortie correspondent respectivement au début et à la fin de l'exécution de la transition de substitution du réseau de niveau supérieur.
La **substitution de transition** permet de remplacer une transition de substitution d'un réseau composite par une méta-transition. Il en résulte un second réseau composite obtenu par l'application de l'algorithme suivant :
– ajout des nœuds de la méta-transition dans le réseau composite
– chaque arc dirigé vers la transition de substitution est remplacé par un arc dirigé vers la transition d'entrée de la méta-transition (et étiqueté par la même fonction de couleur)
– chaque arc partant de la transition de substitution est remplacé par un arc partant de la transition de sortie de la méta-transition (et étiqueté par la même fonction de couleur)
– suppression de la transition de substitution
La **fusion de places** permet de regrouper des places présentes initialement dans différents réseaux composites et qui modélisent la même ressource. La fusion s'effectue en ne gardant qu'une des places que l'on souhaite fusionner et en redirigeant vers cette place tous les arcs attachés aux places supprimées. Ceci implique naturellement que les places que l'on souhaite fusionner doivent avoir le même domaine de couleur.
La **fusion de réseaux** est une opération appliquée à deux réseaux composites. Elle consiste simplement à regrouper les nœuds des deux réseaux.
Enfin, le passage d'un réseau composite à un réseau de Petri coloré se fait en appliquant successivement les opérations de fusion et de substitution jusqu'à ce que le réseau ne contienne ni transition de substitution ni réseau composite.

Nous souhaitons illustrer ces différents concepts à l'aide d'un exemple. Nous allons pour cela concevoir de façon modulaire un réseau modélisant un échange synchrone selon le mode client-serveur. Un client envoie continuellement des requêtes à un serveur. Après l'envoi d'une requête le client se met en attente de la réponse par le serveur puis repasse dans l'état inactif. Le serveur quant à lui se met en attente des requêtes qu'il acquitte immédiatement.

La figure 7.2 est une représentation graphique du réseau composite modélisé. Les transitions de substitution sont représentées par des traits en pointillés. Les boîtes aux angles arrondies représentent des réseaux composites ou des méta-transitions. Le réseau principal, nommé `main`, est constitué de deux modules : le sous-réseau `client` et le sous-réseau `server` qui modélisent respectivement le comportement du client et du serveur. Au niveau du réseau `client` nous modélisons l'envoi d'un message par une transition de substitution : à ce niveau d'abstraction nous ne nous soucions pas de l'implantation de l'envoi. De même la réception et le traitement de la requête par le serveur sont modélisés par la transition de substitution `Treat request` du module `server`. La dernière étape de la conception consiste à décrire les transitions de substitution `Send request` et `Treat request` à l'aide de méta-transitions.
Nous pouvons transformer ce réseau composite en un réseau coloré standard. Nous appliquons pour cela la séquence d'opérations suivante :

1. substitution des transitions de substitution `Send request` et `Treat request` par les méta-transitions correspondantes.
2. fusion des réseaux `Client` et `Server`
3. fusion des places `ack` et `request`

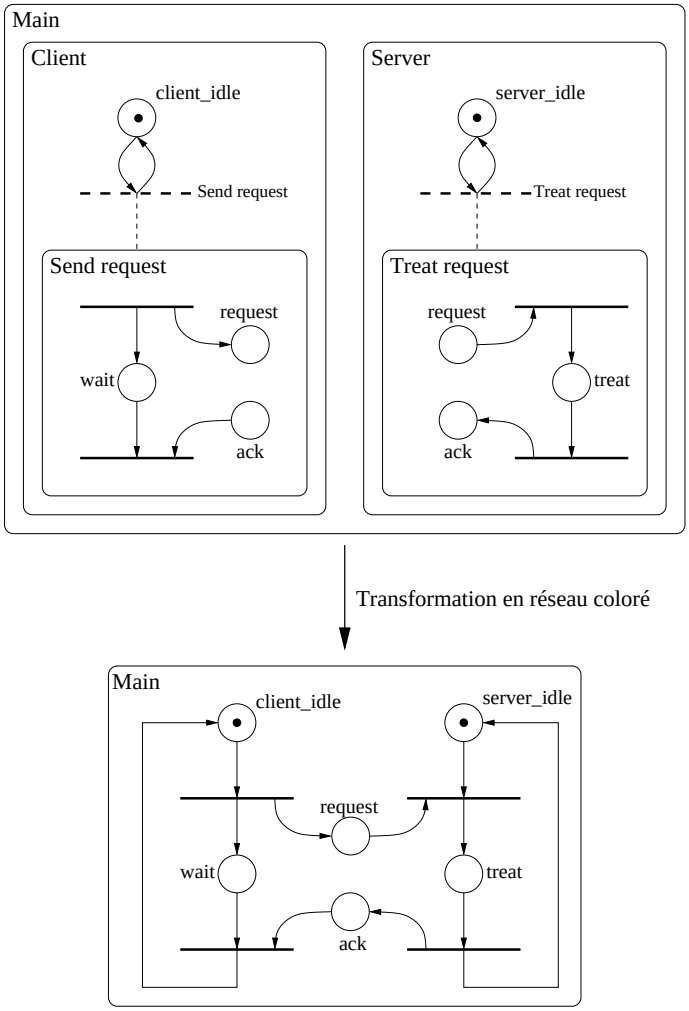


Fig. 7.2 – Un protocole client-serveur modélisé de façon modulaire avec des réseaux hiérarchiques

7.3.2 Exemples de patrons

Quasar associe à chaque construction du langage Ada un réseau composite prédéfini puis parcourt l'arbre syntaxique et traduit chaque élément rencontré par le patron qui lui est associé. Nous allons illustrer ce mécanisme à l'aide de plusieurs exemples de patrons.

a. La tâche

Lors de la conception de Quasar, le besoin d'identifier les tâches du programme apparut rapidement nécessaire. Par exemple, durant un rendez-vous entre deux tâches, la tâche appelante doit identifier la tâche serveur qui doit à son tour mémoriser l'identifiant de la tâche cliente afin de l'acquitter une fois le rendez-vous terminé (cf. le paragraphe plus bas sur les rendez-vous). Ainsi, avant de procéder à la génération du réseau, Quasar parcourt le programme et attribue un identifiant unique à chaque tâche. Il faut pour cela que le nombre de tâches du programme puisse être déterminé statiquement.
La figure 7.3 présente le patron de la tâche. Les places du réseau correspondent aux divers états par lesquels passe toute tâche du programme. Chaque transition a une variable `tid` qui correspond à l'identifiant de la tâche qui exécute la transition. La transition de substitution `declarations` sera substituée par la méta-transition correspondant à la partie déclarative de la tâche. Une fois ces déclarations effectuées, la tâche est prête à exécuter la séquence d'instructions se trouvant dans le corps de la tâche et qui est modélisée par la transition `instructions`. Si la tâche termine correctement elle arrive dans l'état `t.end`. La dernière transition de finalisation permet de libérer la mémoire allouée pour la tâche. Cela consiste, par exemple, à retirer les jetons présents dans les places qui modélisent les variables de la tâche.

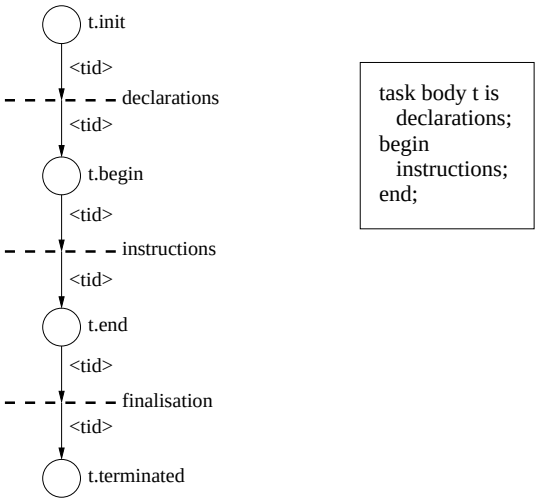


FIG. 7.3 – Patron de la tâche

b. La procédure protégée

Les objets protégés sont des structures de données partagées par les tâches du programme disposant de points d'entrée : les fonctions, procédures et entrées protégées. Afin de résoudre le problème du nommage de ces objets, p.ex., lors d'un appel d'une entrée protégée, Quasar calcule pour chaque objet du programme un identifiant unique. Là encore, il est nécessaire que le nombre d'objets puisse être déterminé statiquement.
Lors de la traduction d'un type ou objet protégé `prot`, les verrous en écriture et en lecture sont modélisés par deux places : la place `prot.rw` et la place `prot.r`. Le domaine de la première place

est réduit à un unique type : le type des objets protégés. Ainsi un jeton `<o>` sera présent dans cette place uniquement si le verrou en écriture de l'objet est libre, i.e., aucune tâche n'exécute une procédure ou une entrée de l'objet. Pour le verrou en lecture nous devons aussi mémoriser le nombre de tâches qui exécutent actuellement une fonction protégée de l'objet. C'est pourquoi le domaine de la place `prot.r` est le produit cartésien du type des objets protégés et des entiers naturels. Ainsi un jeton `<o,n>` présent dans la place `prot.r` exprime le fait que `n` tâches exécutent actuellement une fonction protégée de l'objet `o`.
Le patron de la procédure protégée est décrit par la figure 7.4. La place `proc.call` contient l'ensemble des appels de la procédure `proc`. Elle contient des jetons typés par le produit cartésien du type tâche et du type objet protégé. Ainsi un jeton `<tid,oid>` est présent dans cette place si la tâche identifiée par `tid` applique actuellement la procédure `proc` sur l'objet protégé identifié par `oid`. Pour entrer dans la procédure, deux conditions doivent être remplies :
– Le verrou en écriture doit être libre. Ce test est modélisé par l'arc allant de la place `prot.rw` à la transition `enter` étiqueté par la fonction de couleur `<oid>`, `oid` étant l'identifiant de l'objet sur lequel la procédure est appelée.
– Le verrou en lecture doit être libre. Ce test est modélisé par l'arc aller-retour entre la place `prot.r` et la transition `enter` étiqueté par la fonction de couleur `<oid,0>`. On vérifie ainsi qu'aucune tâche n'exécute actuellement une fonction protégée de l'objet `<oid>`.
Si ces deux conditions sont remplies, la tâche peut entrer dans la procédure protégée et prendre le verrou en écriture en retirant le jeton `<oid>` de la place `prot.rw`. Le corps de la procédure protégée, modélisée par la transition de substitution `body`, peut alors être exécuté. Notons qu'il est nécessaire que la tâche mémorise l'identifiant de l'objet dans lequel elle se trouve afin de libérer le verrou en écriture. Cette action est modélisée par la transition `leave` qui dépose le jeton `oid` dans la place `prot.rw`.

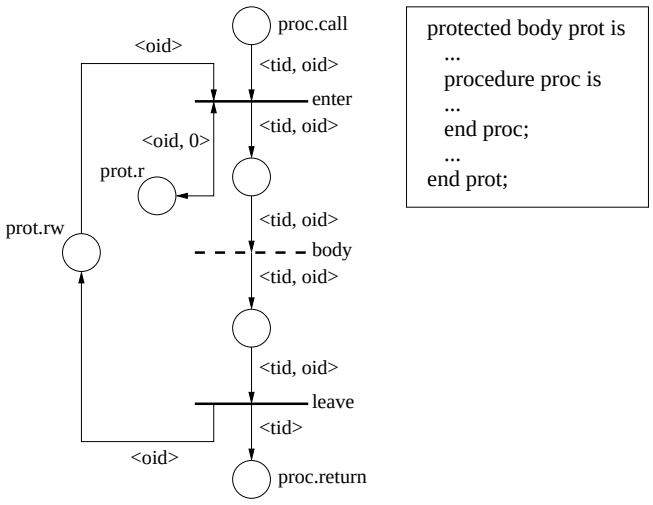


FIG. 7.4 – Patron de la procédure protégée

c. Le rendez-vous

Deux tâches entrent en rendez-vous lorsque une de ces tâches, la tâche cliente, appelle une entrée de l'autre tâche, et que cette seconde tâche, la tâche serveur, exécute une instruction **accept** sur l'entrée appelée. Le rendez-vous consiste en l'exécution par la tâche serveur de la séquence d'instructions se trouvant dans l'instruction **accept** puis par l'acquittement de la tâche cliente. Toutes deux peuvent alors reprendre leurs exécutions.
Il est aisé de modéliser ce mécanisme de synchronisation à l'aide des réseaux colorés. La figure

7.5 présente le patron du rendez-vous. Côté client, la première étape consiste à identifier la tâche serveur (transition `server`). L'appel est alors réalisé en déposant un jeton contenant l'identifiant du client (`tid`) et l'identifiant du serveur (`server`) dans la place `e.call`. Le serveur peut accepter le rendez-vous si un jeton contenant son identifiant est présent dans cette place (transition `accept`). Il exécute ensuite la séquence d'instructions contenue dans le **accept** (transition `instructions`) puis acquitte le client (transition `return`). Afin d'acquitter le client il est nécessaire que le serveur mémorise son identifiant. Ceci est modélisé par l'ajout de la variable `client` au jeton de la tâche. Après réception de cet acquittement (transition `receive`), le client peut poursuivre son exécution. Notons que les deux places nommées `e.call` et les deux places nommées `e.return` seront fusionnées lors de la transformation du réseau composite global en réseau coloré. Cette fusion produira un réseau respectant la sémantique du rendez-vous Ada.

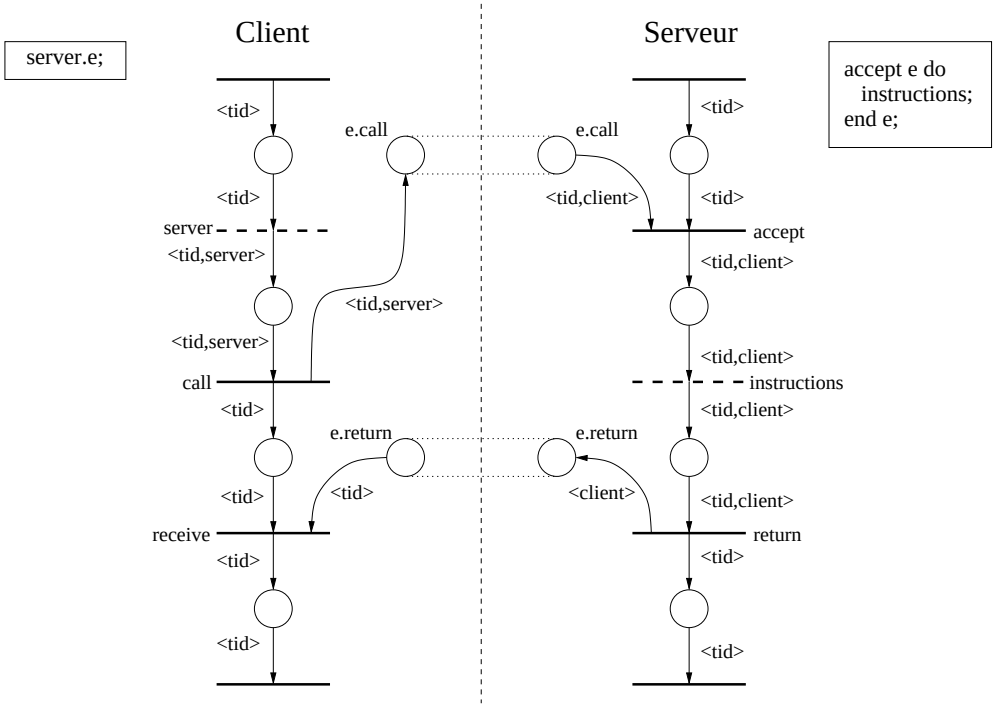


Fig. 7.5 – Patron du rendez-vous

7.4 Modélisation de tâches Ada dynamiques

La modélisation des tâches Ada créées dynamiquement fut l'un des premiers travaux effectué lors de cette thèse [112, 121]. Cette extension apportée à Quasar montre qu'il est possible de modéliser de façon assez efficace les aspects dynamiques des langages de programmation.

7.4.1 Les tâches Ada

Comme nous venons de le voir, dans le langage Ada, un processus séquentiel est appelé tâche. La concurrence entre les tâches peut être obtenue *via* des mécanismes de rendez-vous, d'objet protégé ou encore de variables globales. Tous ces mécanismes sont propres au langage lui-même et possèdent ainsi une sémantique précise.
Pour effectuer l'analyse de programmes concurrents, Quasar modélise le comportement des tâches ainsi que leurs interactions. Nous commencerons alors par expliquer les mécanismes de synchronisation et de dépendances entre les tâches puis nous présenterons les solutions apportées pour la

modélisation de ces dépendances. Une description plus approfondie de ces dépendances peut être trouvée dans [133], [134], [135], et naturellement dans le manuel de référence d'Ada [136].

a. Déclaration de tâche

Une déclaration de tâche est constituée de deux parties : l'interface (ou spécification) et le corps de la tâche (ou *body*).
Le corps de la tâche correspond au code exécuté par la tâche lors de l'exécution. L'interface correspond à la partie visible de la tâche. Elle est également utilisée pour spécifier les entrées de la tâche.

b. Création de tâche

Une tâche peut être créée de deux façons. La première consiste en une création statique : un type de tâche est déclaré puis une variable de ce type est déclarée. La seconde consiste en une création dynamique *via* l'utilisation de pointeurs (*access types*).
Le programme présenté à la figure 7.6 illustre ces deux méthodes de création de tâches.
– la ligne 2 illustre une déclaration d'un type de tâche ;
– un instance statique est déclarée à la ligne 20 ; une telle tâche est alors appelée tâche statique ou tâche élaborée ;
– la ligne 10 illustre la création dynamique de tâche ; le type de pointeur est déclaré à la ligne 3 et le pointeur à la ligne 7 ; une telle tâche est appelée tâche dynamique ou tâche allouée ; l'instruction *new* fait référence à l'allocateur de mémoire utilisé pour la création de la tâche.

c. Hiérarchie entre les tâches

Le langage Ada est un langage structuré en blocs dans lequel des blocs peuvent être imbriqués au sein d'autres blocs.
Une tâche peut être déclarée dans un bloc quelconque créant alors une hiérarchie entre les tâches. Ainsi, une tâche directement responsable de la création d'une autre tâche est appelée parent de la tâche alors que la tâche créée est considérée comme fille de sa tâche parente.
La création et la terminaison d'un tâche modifie le comportement des autres tâches dans la hiérarchie des tâches. Le parent d'une tâche fils est responsable de la création de cette tâche.
Les dépendances entre les tâches sont basées sur la notion de maître d'une tâche présentée dans [135] :

[**Master 1.**] Le parent d'une tâche élaborée est également le maître direct de cette tâche.

[**Master 2.**] Une instance qui déclare un type de pointeur de tâche est le maître direct de toutes les instances de tâches allouées créées par ce type de pointeur.

[**Indirect Master.**] Une instance est appelée maître indirect d'une instance de tâche si (1) elle invoque – directement ou indirectement – un maître de la tâche ou (2) elle est maître d'un maître de l'instance.

Toute instance de tâche– sauf la tâche principale – possède alors exactement un maître direct et potentiellement plusieurs maîtres indirects.
Le maître d'une tâche doit alors attendre que cette tâche se termine avant de pouvoir terminer lui-même. Le parent d'une tâche est souvent le maître de cette tâche, cependant, dans le cas de tâches dynamiques, le maître d'une tâche n'est pas son parent mais la tâche contenant la déclaration du type de pointeur.

Pour illustrer ces notions, considérons à nouveau la figure 7.6. La première règle implique que le parent de la tâche `T` élaborée à la ligne 20 est également son maître. Comme le parent de cette tâche est la procédure `ALLOC`, il est aussi son maître.

```
1 procedure ALLOC is
2    task type TT;
3    type OUTT is access TT;
4
5    procedure P is
6       type INTT is access TT;
7       O : OUTT;
8       I : INTT;
9    begin -- body of P
10      O := new TT; -- allocated / dynamic task
11      I := new TT; -- allocated / dynamic task
12      ...
13      -- procedure P completion
14   end P;
15
16   task body TT is
17      ...
18   end TT;
19
20   T : TT; -- elaborate / static task
21
22 begin -- body of ALLOC
23   P; -- call P
24   ...
25   -- ALLOC completion
26 end ALLOC;
```

FIG. 7.6 – Déclaration et création de tâches

La seconde règle implique que, même si le pointeur `O` est alloué par la procédure `P` à la ligne 10, puisque le type de pointeur `OUTT` est déclaré par la procédure `ALLOC`, cette procédure est le maître de la tâche créée à la ligne 10.
Enfin, la troisième règle implique que `ALLOC` est un maître indirect de la tâche `I`, puisqu'elle invoque `P` qui est un maître direct de `I`.

d. Activation, exécution, finalisation et terminaison de tâche

Nous pouvons désormais présenter les synchronisation inter-tâches induites par ces dépendances pour chaque étape du cycle de vie d'une tâche.
- *Activation.* Cette étape correspond à l'élaboration de la partie déclarative de la tâche, la création et l'initialisation des variables locales. Une tâche ne peut accomplir sa phase d'activation avant que toutes la tâches déclarées dans sa partie déclarative n'aient terminé leur propre phase d'activation. C'est le premier point de synchronisation. L'activation d'un fils d'une tâche démarre à la fin de l'élaboration de son parent.
  Notons que dans le cas d'une tâche allouée, le parent est bloqué sur l'appel à l'allocateur jusqu'à ce que l'instance ait terminé sa phase d'activation.
  Notons enfin qu'une entrée de tâche peut être appelée avant que cette tâche ait été activée.
- *Execution.* Cette étape correspond à l'exécution du corps de la tâche. Aucune synchronisation n'est introduite lors de cette phase.
- *Completion.* Cette phase correspond à l'état dans lequel se trouve une tâche une fois que toutes les instructions de son corps ont été exécutées. Une tâche ne peut sortir de cet état et démarrer sa phase de terminaison avant que toutes les tâches dépendants d'elle n'aient finies leur phase de terminaison. C'est le second et dernier point de synchronisation.

- *Termination.* Cette étape correspond à la destruction de la tâche. Là encore, aucune synchronisation n'est introduite.

7.4.2 Modélisation de tâche

Nous présentons ici comment modéliser une tâche Ada ainsi que ses mécanismes internes de synchronisation en un réseau de Petri coloré.
Pour modéliser correctement le comportement des tâches, nous sommes obligé de redéfinir le patron associé à une tâche. Celui est désormais composé des 4 "meta-transitions" correspondant aux 4 étapes du cycle de vie et 5 places intermédiaires (places `C.Begin`, `C.Ready`, ..., `C.End`). Ce patron est présenté sur la figure 7.7.
Pour modéliser les dépendances entre tâches, plusieurs problèmes ont besoin d'être résolus : le nommage dynamique de tâche, le référencement dynamique des maîtres et la modélisation des synchronisations liées aux dépendances.

a. Nommage dynamique

Comme nous l'avons vu précédemment, dans Quasar, chaque tâche doit posséder un identifiant unique. Lorsque toutes les tâches sont des tâches élaborées, le nombre de tâches est alors constant et connu à la compilation. Les identifiants peuvent alors être assignés statiquement et restent les même pour n'importe qu'elle exécution du programme.
Lorsque le nombre de tâches est variable et peut changer d'une exécution à une autre, le réseau de Petri doit être capable de générer automatiquement et dynamiquement des identifiants uniques. Pour éviter d'ajouter de la combinatoire en rendant la génération d'identifiant totalement indéterministe, l'objectif est alors de trouver un moyen de générer des identifiants qui changeront le moins possible – et idéalement jamais – d'une exécution à une autre.
Trois solutions ont été considérées pour gérer le nommage dynamique. Nous illustrerons ces solutions en utilisant le petit programme présenté à la figure 7.8.

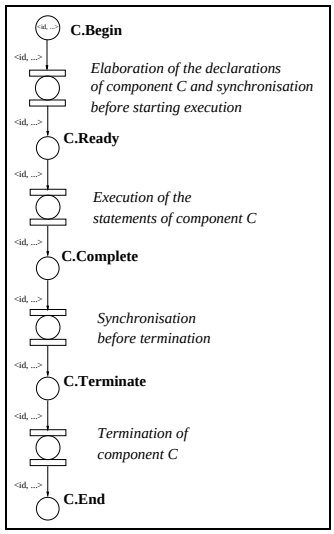


FIG. 7.7 – Modèle d'un composant dans Quasar

i. Serveur d'identification global

Une première solution consiste à utiliser un serveur global d'identifiant qui assignera à chaque tâche créée un identifiant unique. Le principal désavantage lié à cette solution est la combinatoire introduite par l'attribution des identifiants. En effet, pour chaque exécution d'un programme, une tâche peut obtenir différents identifiants. Comme toutes les exécutions possibles sont considérées lors de l'exploration, le combinatoire introduite peut être gigantesque.
Pour illustrer cette solution, le réseau de Petri coloré correspondant au programme de la figure 7.8 est présenté à la figure 7.9. Ce réseau est simplifié puisque seul le mécanisme d'identification nous intéresse véritablement ici. Les tâches élaborées aux lignes 25, 26 et 27 sont représentées par les jetons dans les places `T1.BEGIN` et `T2.BEGIN`. La première valeur dans le jeton est l'identifiant de la tâche, la seconde représente les différentes valeurs que peut contenir le jeton.
Pour ce simple programme, le graphe généré possède près de 500 états et aboutit à 6 possibilités de nommage (cf. tableau 7.1).

ii. Serveur d'identification local

Une seconde possibilité consiste à gérer plusieurs serveurs locaux. Ici, un serveur spécifique est ajouté pour chaque instruction du programme pouvant générer de nouvelles tâches. Pour éviter les collisions d'identifiants, nous assignons à chaque serveur un identifiant unique. Ici, le nombre d'instructions pouvant générer de nouvelles tâches est constant et connu lors de la compilation, il est alors possible de nommer statiquement chacun des serveurs locaux.

Les identifiants de tâches sont alors composés de deux valeurs : l'identifiant du serveur local ayant identifié la tâche et l'identifiant unique donné par le serveur local. Cette solution permet de limiter les combinaisons à un niveau local et non plus global.
Pour illustrer cette solution, considérons alors le modèle pour le programme de la figure 7.8 obtenu en utilisant les serveurs locaux. Dans ce programme, les lignes 15, 22, 25, 26 et 27 sont des instructions ou des déclarations qui créent des tâches. Le réseau de Petri généré aura alors 5 serveurs locaux. Si l'on considère les tâches `T1`, `T2` and `T3`, le réseau généré sera celui présenté à la figure 7.10.
Le premier serveur local correspond à l'instruction `new` de la ligne 15, tandis que le second correspond à l'instruction `new` de la ligne 22. La première valeur du jeton de la tâche représente l'identifiant du serveur, la seconde valeur correspond à l'identifiant obtenu auprès du serveur local. En utilisant les serveurs locaux pour le programme 7.8, on aboutit alors à un peu plus de 200 états,

```
 1 procedure Identification is
 2
 3    task type T3;
 4    task body T3 is
 5    begin
 6       null;
 7    end T3;
 8
 9    type Access_T3 is access T3;
10
11    task type T1;
12    task body T1 is
13       O : Access_T3;
14    begin
15       O := new T3;
16    end T1;
17
18    task type T2;
19    task body T2 is
20       P : Access_T3;
21    begin
22       P := new T3;
23    end T2;
24
25    T1_1 : T1;
26    T1_2 : T1;
27    T2_1 : T2;
28
29 begin
30    null;
31 end Identification;
```

FIG. 7.8 – Un programme Ada simple.

c'est-à-dire plus de 2 fois moins d'états que dans le cas du serveur global. Le nombre de possibilités de nommage est également réduit à 2.

iii. Fonction de nommage injective

Une troisième et dernière solution consiste à utiliser une fonction injective pour effectuer l'attribution des identifiants. Cette fonction est exécutée lors du franchissement de la transition qui crée une nouvelle tâche.
Pour chaque tâche, on conserve une valeur n modifiée à chaque création de tâche (cette valeur peut alors être le nombre de tâches créées ou l'identifiant de la dernière tâche créée). L'identifiant d'une nouvelle tâche est alors obtenu en appelant la fonction injective de nommage avec comme entrées l'identifiant du parent de la tâche ainsi que la valeur n du parent.
Nous avons choisi d'utiliser cette dernière solution comme mécanisme de nommage puisque c'est celle qui génère le moins de combinatoire comme nous pouvons le voir sur le tableau 7.1.

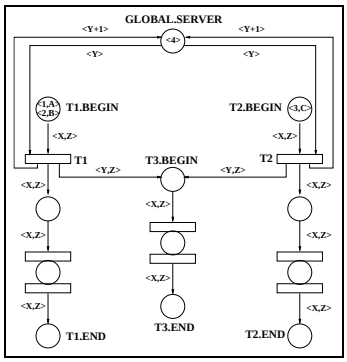


FIG. 7.9 – Partie du réseau généré pour le programme de la figure 7.8 en utilisant le serveur d'identification global.

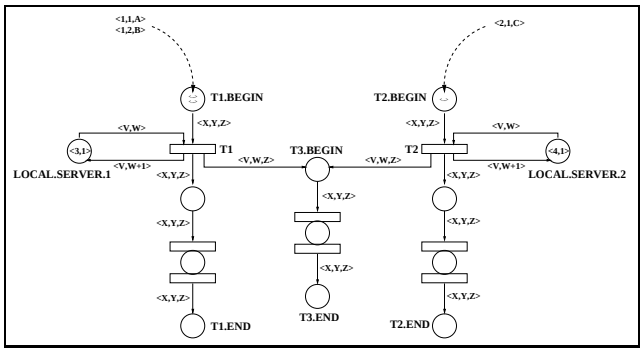


FIG. 7.10 – Partie du réseau généré pour le programme de la figure 7.8 en utilisant le serveur d'identification local.

id allocation mecanims	*Number of generated states*	*Number of different name sequences*
Global *id* server	493	6
Local *id* server	205	2
One-to-one function	125	1

TAB. 7.1 – Evaluation de la combinatoire engendrée pour le programme de la figure 7.8 en utilisant le serveur global, les serveurs locaux et la fonction de nommage injective(figures 7.9, 7.10 et 7.11).

Pour trouver une bonne fonction, nous pouvons alors considérer trois cas :
– si toutes les tâches sont générées par une même instruction, on peut alors utiliser la fonction suivante :

$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(id,n) = id + n + 1$$

où *id* représente l'identifiant du parent de la tâche et n le nombre de tâches déjà créées par ce parent
– si le nombre de tâches que pourra générer le programme peut être borné, on peut alors utiliser la fonction définie par :

$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(id,n) = id + n + max$$

où *id* représente l'identifiant du parent de la tâche, n l'identifiant de la dernière tâche créée par ce parent et *max* le nombre maximale de tâches qui peuvent être créées dans le programme.
– dans les autres cas, il est alors nécessaire de définir une fonction injective spécifique.

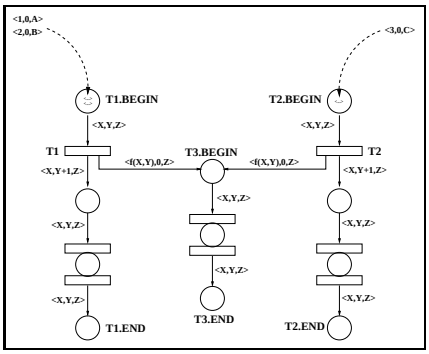


FIG. 7.11 – Partie du réseau généré pour le programme de la figure 7.8 en utilisant la fonction de nommage injective

b. Référencer le maître d'une tâche

Pour des questions de simplicité, nous avons décidé de modéliser différemment les tâches allouées et les tâches élaborées. Un sous-réseau est alors généré pour chaque type de tâche et pour chaque type de pointeur de tâche, un sous-réseau spécifique est généré. Par exemple, le modèle correspondant au programme présenté à la figure 7.6 contiendra trois différents sous-réseaux pour le type `TT` : un pour le type lui-même, un pour le type `INTT` et un dernier pour le type `OUTT`.
Pour pouvoir retrouver le maître d'une tâche, nous devons stocker partiellement la hiérarchie de tâches. Cette hiérarchie est spécifique à chaque bloc qui déclare des types de pointeurs de tâche. Nous avons alors considéré deux possibilités de modélisation de cette hiérarchie : soit en utilisant des places spécifiques, soit en ajoutant de l'information dans les jetons des tâches.

i. Modéliser la hiérarchie entre tâches dans des places spécifiques

Il est possible de sauvegarder l'identifiant du maître d'une tâche allouée dans une place spécifique à chaque type de pointeur de tâche. Cette place contient alors un ensemble de jetons $\langle id, id\_master \rangle$ où id représente l'identifiant de la tâche allouée ou élaborée et id_master l'identifiant du maître du bloc correspondant. Ainsi, chaque fois qu'une tache allouée se termine, elle peut retrouver l'identifiant de son maître *via* cette place spécifique.

Pour le programme 7.6 il faut alors définir deux places : une pour le bloc `Alloc` (que nous noterons p_A) et une pour la procédure `P` (que nous noteronsp_P). Lorsqu'une tâche id_0 (qui peut être la tâche principale) appelle la procédure `Alloc` elle place un jeton $\langle id_0, id_0 \rangle$ dans p_A. La déclaration de `T` aboutit à l'ajout d'un jeton $\langle id\_T, id_0 \rangle$ dans la place p_A (où id_T représente l'identifiant associé à `T`) ; ainsi `T` peut allouer des tâches de type `OUTT` et leur donner le maître correspondant. Lorsque la tâche id_0 appelle P, elle place un jeton $\langle id_0, id_0 \rangle$ dans p_P. L'allocation de `O` ajoute alors le jeton $\langle id\_O, id_0 \rangle$ dans les places p_A et p_P puisque les deux blocs déclarent des pointeurs de tâches visibles par `O`. La même opération est effectuée lors de l'allocation de `I`.
L'avantage de cette solution est de ne pas complexifier le jeton des tâches et ainsi ne pas dégrader les performances des franchissements de transitions. Le principal inconvénient de cette approche est qu'elle complique les modèles de synchronisation et peut alors aboutir à de moins bonnes réductions lors de l'analyse statique du réseau de Petri.

ii. Modéliser la hiérarchie entre tâches dans les jetons

La seconde solution consiste à ajouter un tableau d'identifiant dans chaque jeton de tâche. Ce tableau conserve un historique de l'exécution et permet ainsi aux tâches de retrouver leur maître. La taille des tableaux est fixée lors de la compilation et est fixée au nombre de blocs qui déclarent des types de pointeurs de tâche. Pour chacun de ces blocs qui peut être un maître possible de tâches allouées, une entrée est ajoutée dans le tableau.

Dans le réseau, lorsqu'un bloc déclare un type de pointeur, il place son identifiant dans l'entrée du tableau correspondante. Lorsqu'un bloc crée un nouvelle tâche ou appelle un nouveau bloc, il lui transmet son tableau d'identifiant. Le principe consiste à propager ce tableau afin que les tâches allouées puissent retrouver leur maître et récupérer son identifiant à l'entrée correspondante du tableau.
Pour illustrer ce mécanisme, considérons l'exemple présenté à la figure 7.6. La procédure `ALLOC` déclare un type de pointeur `OUTT`, une entrée est donc ajoutée dans le tableau. La procédure `P` déclare un type de pointeur `INTT`, une seconde entrée est alors ajoutée. Comme il n'y a pas d'autres type de pointeur dans le programme, la taille des tableaux sera de 2 : la première entrée sera pour l'identifiant de la procédure `ALLOC` tandis que la second sera pour l'identifiant de la procédure `P`. Dans le réseau généré, lorsque la procédure `ALLOC` appelle la procédure `P` à la ligne 27, elle lui donne son tableau d'identifiants (c'est-à-dire, $[alloc\_id, null]$). Ainsi, lorsque `P` déclare un type de pointeur à la ligne 6, elle ajoute son identifiant à l'entrée correspondante dans le tableau. Lorsque `P` engendre `O` et `I`, elle leur passe son tableau ($[alloc\_id, p\_id]$). Comme le maître de `O` est la procédure `ALLOC`, `O` a simplement à récupérer le bon identifiant dans l'entrée correspondante du tableau (c'est-à-dire, la première entrée) et comme le maître de `I` est la procédure `P`, `I` a simplement à récupérer l'identifiant correspondant à son maître (c'est-à-dire, le second dans le tableau).
Le principal désavantage de cette solution est que chaque jeton de tâche doit conserver ce tableau même s'il ne l'utilise jamais. De plus, si de nombreux types de pointeurs sont déclarés, la taille du tableau sera d'autant plus grande nécessitant alors plus de mémoire pour la représentation des états. Un bon moyen de réduire cette augmentation est de grouper les déclaration de types de pointeurs dans un faible nombre de blocs. Ceci aboutira alors à une réduction de la taille du tableau.

c. Patterns de gestion des tâches avec hiérarchie dans les places

Une tâche allouée dépend de son maître qui n'est pas nécessairement son parent mais toujours un ascendant. Afin de gérer les dépendances, on utilise alors une place nommée `C.Id_Master` dans le modèle de bloc générique contenant un type de pointeur de tâche. Cette place contient des jetons de type $\langle id, id\_master \rangle$ où id_master référence le maître de la tâche id.

La seconde place introduite est la place `Task_Type_Name.Dependences` qui stocke le nombre de tâches allouées qu'un maître devra attendre lors de sa terminaison. Une place de ce type est alors générée pour chaque type de pointeur de tâche. Cette place contient des jetons $\langle id\_master, cpt \rangle$ où id_master représente l'identifiant de la tâche qui a élaborée le bloc contenant la déclaration du type de pointeur et cpt sert à récupérer le nombre de tâches de ce type qui ont été activées.
Lorsqu'une tâche id (qui peut être la tâche principale) élabore un composant `C` contenant la définition d'un type de pointeur de tâche (cf. figure 7.18, transition t1) un jeton $\langle id, 0 \rangle$ est déposé dans la place `Task_Type_Name.Dependences` où `Task_Type_Name` est le nom du type. Cette transition initialise le compteur associé à ce type en notant qu'à cette étape, `C` ne dépend d'aucune tâche.
Le franchissement de cette transition est également utilisé pour sauvegarder le fait que la tâche id est le maître de ce type en ajoutant un jeton $\langle id, id \rangle$ dans la place `C_Name.Id_Master`. La tâche qui a élaboré le composant C ne peut alors atteindre la place `C.Terminate` tant que le nombre de tâche actives dépendant d'elle n'est pas nul (transition t2).

Lorsqu'une tâche– allouée ou élaborée– est créée, elle doit connaître les maître de chacun des type de pointeur de tâche visible (que l'on connaît dès la phase de compilation). Pour cela, nous utilisons le pattern suivant qui prépare l'allocation ou la déclaration (figure 7.13).
Son rôle est double :
– premièrement, il calcule un nouvel identifiant (`id_child`, transition t1) pour la nouvelle instance de tâche.
– deuxièmement (transition t2), il note que cette nouvelle tâche peut allouer des tâches de chaque type de pointeur de tâche visible. Ainsi, un jeton $\langle id, id_i \rangle$ est placé dans chaque place `Ci.Id_Master`

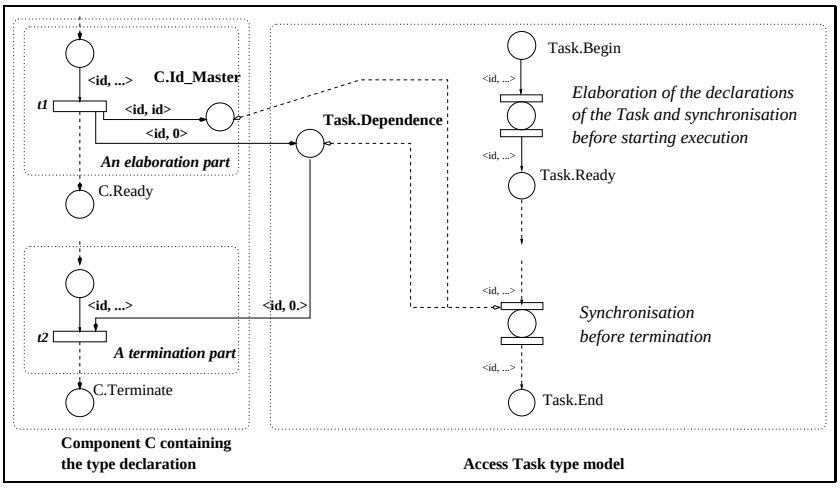


FIG. 7.12 – Elaboration d'un composant contenant une définition d'un type de pointeur de tâche.

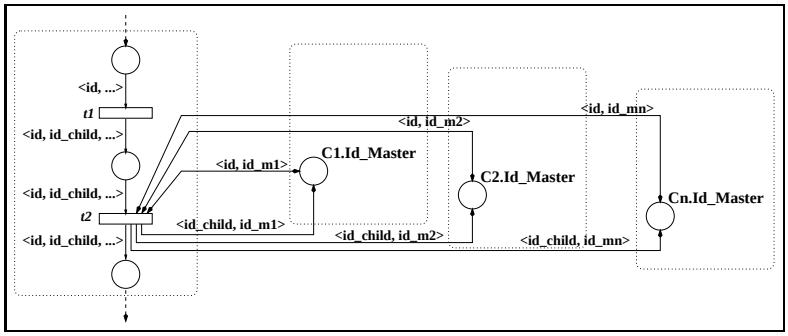


FIG. 7.13 – Préparation de la création d'une tâche dynamique

où `Ci` représente le bloc déclarant de type de pointeur de tâche visible par la tâche créée. Le maître de ce composant est connu grâce au jeton $\langle id, id_i \rangle$ présent initialement dans cette place.

A l'inverse, lors de la terminaison, tous les jetons placés dans les places `Ci.Id_Master` pendant la préparation de la création doivent être retirés. C'est le rôle du pattern présenté à la figure 7.14.

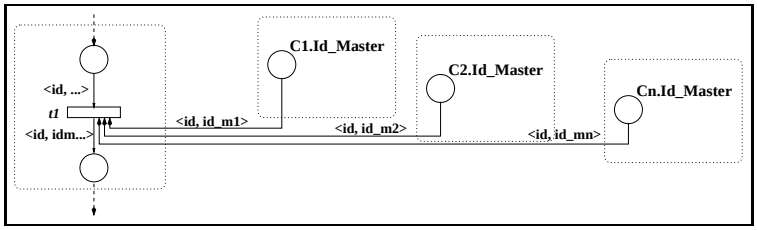


FIG. 7.14 – Préparation de la terminaison

Pour l'allocation d'une nouvelle tâche (voir figure 7.19), il faut tout d'abord préparer la création en utilisant le pattern décrit précédemment à la figure 7.13. Puis (transition t1), le nombre de tâche dont dépend le bloc qui a élaboré le type de la tâche allouée (par l'exécution de la tâche *id_master*) est incrémenté en modifiant le jeton $\langle id\_master, cpt \rangle$ en $\langle id\_master, cpt + 1 \rangle$. La nouvelle tâche est alors démarrée (transition t2) en déposant un jeton $\langle id, id\_child \rangle$ dans la place `T.Begin`. Enfin (transition t3) la tâche parent doit attendre la fin de l'élaboration de la tâche allouée pour continuer. La terminaison d'une tâche allouée consiste à préparer la terminaison en utilisant le pattern pré-

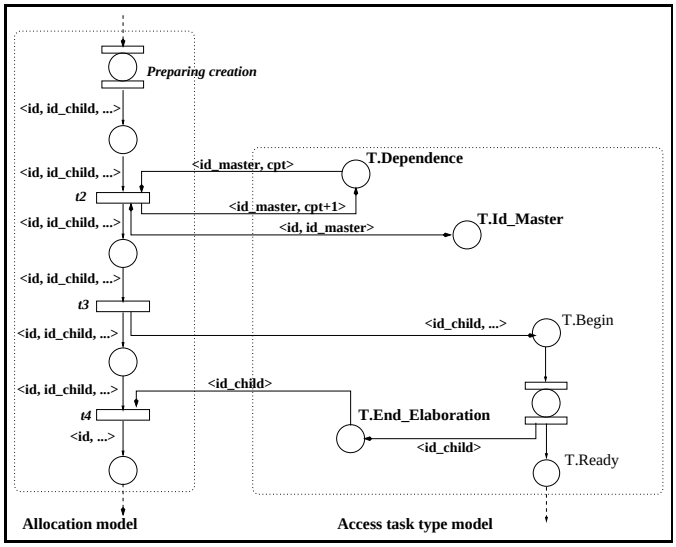


FIG. 7.15 – Evaluation d'un appel à l'allocateur de tâche

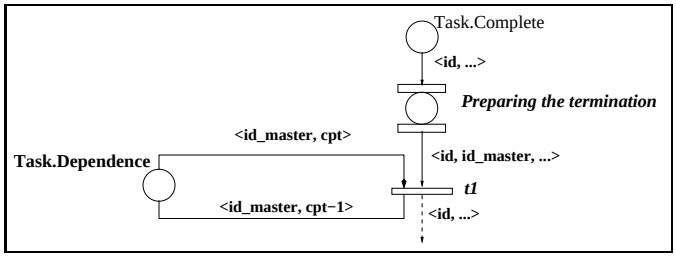


FIG. 7.16 – Terminaison d'une tâche allouée

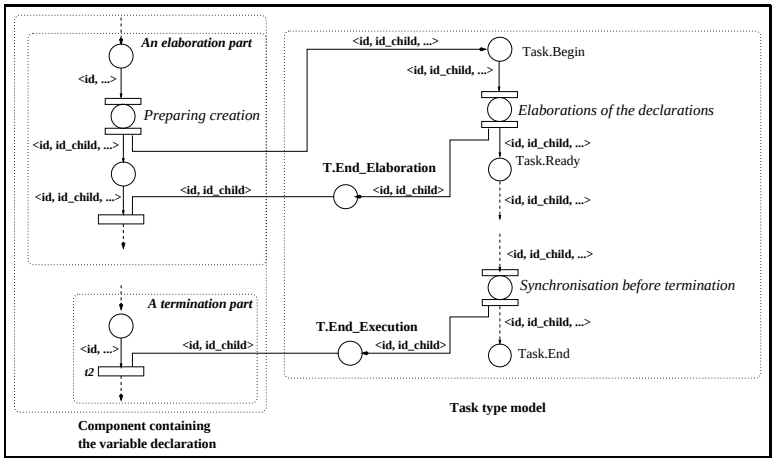


FIG. 7.17 – Création d'une tâche par évaluation d'une déclaration

sentée sur la figure 7.16. puis à décrémenter le compteur associé à ce type de tâche allouée.
La création d'une tâche par évaluation d'une déclaration est modélisée par le pattern présenté à la figure 7.20. Dans un premier temps, la création est préparée comme précédemment. Puis la tâche est activée et son parent attend la fin de son élaboration. Lors de sa terminaison, la tâche créée place un jeton dans la place `T.End_Execution`. Ce jeton permet alors à son parent de continuer (transition t2).

d. Patterns de gestion des tâches avec hiérarchie dans les jetons

Nous définissons maintenant les patterns utilisés pour la génération de réseaux de Petri colorés avec gestion des tâches dynamiques modélisant la hiérarchie des tâches directement dans les jetons et non plus dans des places spécifiques.
Le premier pattern présenté sur la figure 7.18 concerne l'élaboration d'un bloc contenant une définition d'un type de pointeur de tâche.
Il fonctionne de la façon suivante : lorsqu'une tâche *id* (qui peut être la tâche principale) élabore un composant contenant la définition d'un type de pointeur de tâche, un jeton $\langle id, 0\rangle$ est déposé dans la place `Task_Type_Name.Dependences` où `Task_Type_Name` représente le nom du type. Cette transition initialise le compteur associé à ce type en notant que, à cette étape, `C` ne dépend d'aucune tâche. La tâche qui a élaboré le composant C ne peut alors atteindre la place `C.Terminate` tant que le nombre de tâche actives dépendant d'elle n'est pas nul (transition t2).

Pour l'allocation d'une nouvelle tâche, nous utilisons le pattern présenté sur la figure 7.19. Dans un premier temps, le nombre de tâches duquel le bloc qui a élaboré le type de la tâche allouée dépend puis le jeton $\langle id\_master, cpt\rangle$ est modifié en $\langle id\_master, cpt + 1\rangle$ dans la place `Task.Dependence`. La transition t2 est ensuite franchie et créée une nouvelle tâche en déposant un jeton $\langle id, id\_child\rangle$ dans la place `T.Begin` (l'identifiant de la nouvelle tâche est obtenu de la même manière que dans la section précédente). Enfin, avant de franchir la transition t3, la tâche attend que la tâche allouée ait terminé son élaboration. La place $V.Var$ est utilisée pour conserver que la tâche *id* a alloué la tâche *id_child* ce qui permettra la synchronisation lors de la terminaison.
La création d'une tâche par évaluation d'une déclaration est effectuée en utilisant le pattern présenté à la figure 7.20. La tâche créée est activée et le parent attend la fin de son élaboration. Lors de sa terminaison, la tâche créée signale sa terminaison en déposant un jeton dans la place `T.End_Execution`. Ce jeton permet à la tâche parent de continuer (transition t2).

e. Evaluations

Nous avons évalué les deux approches pour la modélisation de la hiérarchie des tâches. Nous avons utilisé pour cela le modèle du modèle client/serveur(figure 7.21). Les résultats sont reportés sur le tableau 7.2.

	Token *version of* QUASAR		**Places** *version of* QUASAR	
Clients	*States without reductions*	*States with Reductions*	*States without reductions*	*States with Reductions*
1	2 549	247	2 699	246
2	438 913	9 499	536 335	12 357
3	–	735 767	–	1 347 009

TAB. 7.2 – Evaluation des deux méthodes de modélisation des dépendances pour le programme client/serveur.

On observe alors que la modélisation de la hiérarchie dans les jetons aboutit à une combinatoire plus faible que ce soit avec ou sans les différents types de réductions (réductions structurelles et réductions lors de l'exploration). Ces résultats peuvent être en partie expliqués par la préparation

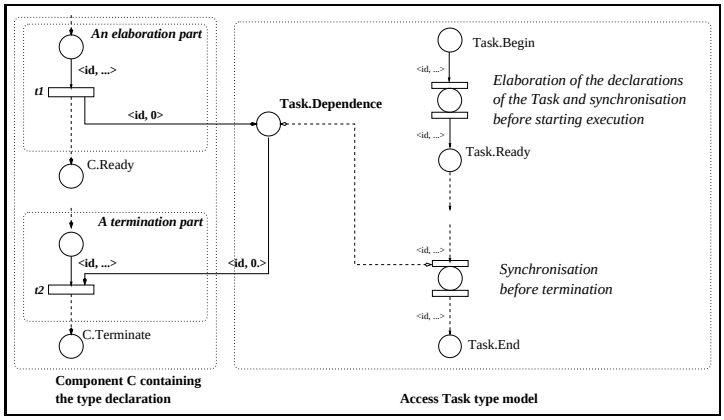


FIG. 7.18 – Elaboration d'un composant contenant une définition d'un type de pointeur de tâche

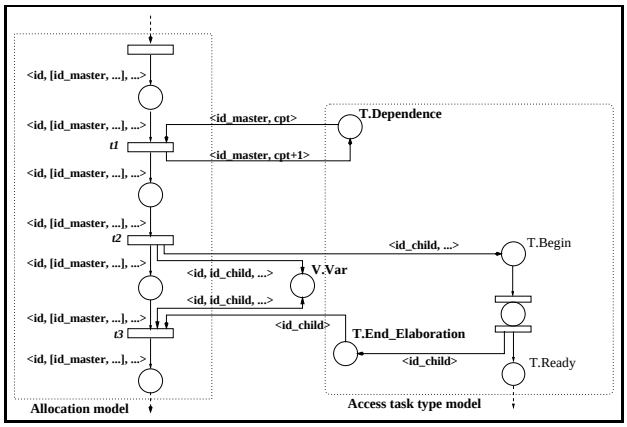


FIG. 7.19 – Evaluation d'un allocateur de tâche

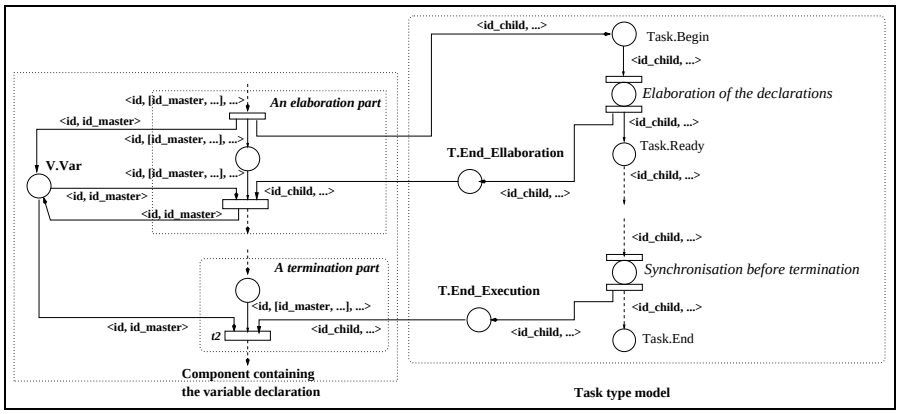


FIG. 7.20 – Création d'une tâche par évaluation d'une déclaration

```
procedure Server is
   Max_Client : Integer := 5;

   protected Data is                    -- The accessed data
      procedure Get_Value (Value : out Integer);
   private
      Data_Value : Integer := 0;
   end Data;

   protected body Data is
      procedure Get_Value (Value : out Integer)is
      begin
         Data_Value := Data_Value + 1;
         Value := Data_Value;
      end Get_Value;
   end Data;

   task type Thread is                  -- Only the threads can access the data
      entry Get_Value (Param : out Integer);
   end Thread;
   type Access_Thread is access Thread;

   task body Thread is
   begin
      accept Get_Value (Param : out Integer) do
         Data.Get_Value(Param);
      end Get_Value;
   end Thread;

   task type Task_Server is
      -- The task server, create a thread for each client's request
      entry Get_Thread(Id : out Access_Thread);
   end Task_Server;

   task body Task_Server is
   begin
      for I in 1..Max_Client loop
         accept Get_Thread (Id : out Access_Thread) do
            Id := new Thread;
         end Get_Thread;
      end loop;
   end Task_Server;

   The_Task_Server : Task_Server;       -- The task server

   task type Client;                    -- Client of the task server
   type Access_Client is access Client;

   task body Client is
      Id    : Access_Thread;
      Value : Integer;
   begin
      The_Task_Server.Get_Thread(Id); -- Get the thread
      Id.Get_Value(Value);            -- Get the value of the data
   end;

   A_Client : Access_Client;

begin
   for I in 1..Max_Client loop          -- main loop, creates clients
      A_Client := new Client;
   end loop;
end Server;
```

FIG. 7.21 – Un programme client/serveur

de la terminaison dans le cas de la modélisation de la hiérarchie dans les places. Comparée à la seconde approche, elle ajoute un transition supplémentaire dans le modèle qui peut alors générer de la combinatoire supplémentaire.

7.4.3 Exemples

Dans ces exemples, on nomme les tâches en utilisant une fonction de nommage injective et l'on modélise la hiérarchie entre les tâches dans les jetons.

a. Le crible d'Eratosthene

Le crible d'Eratosthene est utilisé pour trouver tous les nombres premiers inférieurs à un nombre N. Il considère chaque nombre entre 2 et N et élimine tous les multiples de chaque nombre considéré. Nous avons implémenté une version concurrente du crible avec création dynamique de tâches présenté à la figure 7.22.

Pour chaque nombre premier, une nouvelle tâche est générée. Le programme peut être vu comme un *pipeline* de tâches dynamique puisqu'une tâche peut être ajouté directement au *pipe*. Chaque étage du *pipe* est donc composé d'un nombre premier associé à une tâche qui représente une partie du crible. Comme on peut le voir sur l'algorithme 7.22, la procédure principale est une simple boucle qui appelle l'entrée de la première tâche `Prime` en lui transmettant chacun des nombres à passer au crible. A la fin de la boucle, elle diffuse un message de terminaison dans le *pipe*.
Le corps des tâches `Prime` est constitué d'une simple boucle qui itère jusqu'à ce que le message de terminaison soit reçu. La tâche attend sur son entrée `Test_Primality`, sauvegarde la valeur reçue à la fin de l'appel. Puis, la tâche teste la primalité du nombre reçu ; si la tâche n'a pas encore stocké un nombre premier, elle conserver le nombre reçu en tant que nombre premier et se remet en attente sur son entrée. Dans le cas où la tâche a déjà stocké un nombre premier, elle test si le nombre reçu est divisible par son nombre premier. Si c'est le cas, le nombre n'est pas premier. Sinon, elle transmet la valeur à sa tâche voisine (en la créant si nécessaire *via* la fonction `Create_Prime`). Dans cet exemple, nous ne récupérons pas les nombres premiers car seul l'aspect dynamique de création et de terminaison de tâche nous intéresse.
Les résultats des expérimentations sont présentés dans le tableau 7.3.

N	*Running tasks*	*States without reductions*	*States with reductions*	*Reduction factor*
3	3	1 556	294	5
6	4	19 160	1 784	11
10	5	224 102	10 047	22
12	6	2 810 870	65 645	43

Tab. 7.3 – Expérimentations pour le crible d'Eratosthene

En fonction de la valeur de N, le nombre de tâches exécutées est reporté. On reporte également la taille du graphe d'états avec et sans réductions (là encore réductions au sens large, c'est-à-dire tant structurelles que dynamiques, i.e. lors de l'exploration).

b. Le programme Client/Serveur

Le programme présenté à la figure 7.21 décrit le comportement d'une application client/serveur simple. La boucle principale crée un certain nombre de clients qui envoient des requêtes sur le serveur. Pour chaque requête reçue, le serveur crée un `Thread` en allouant un pointeur `Access_Thread` et en retournant ce pointeur au client. Le client peut alors appeler l'entrée du `Thread` qui accède alors à l'objet protégé représentant la donnée critique.

Pour différents nombres de clients, nous avons reporté la taille du graphe d'accessibilité là encore avec et sans réductions. Les résultats sont présentés sur le tableau 7.4.

Clients	*Running tasks*	*States without reductions*	*States with reductions*	*Slicing & reductions*
1	4	2 549	247	221
2	6	438 913	9 499	5 939
3	8	–	735 767	239 723
4	10	–	–	12 847 017

TAB. 7.4 – Expérimentations pour le programme client/serveur

Les résultats obtenus sur ces deux modèles montrent bien que, pour les tâches allouées, l'introduction des mécanismes de synchronisation génère une certaine combinatoire. Cependant, les réductions utilisées à chaque étape du processus suivi par Quasar permettent de limiter cet ajout et de réduire fortement la taille du graphe.

7.4.4 Remarque

Modifions alors le programme précédent de manière à y introduire une erreur non détectable à la compilation mais qui rend le programme incorrect dans certains cas.
Une exécution sans *deadlock* aboutit, dans le cas général, à deux comportements possibles : soit on aboutit à des séquences finies où les états terminaux (i.e, pour lesquels plus aucune transition n'est franchissable) sont des états où seules les places `END` sont marquées car tous les processus se sont terminés correctement, soit on aboutit à des séquences infinies mais dont le nombre d'états est fini (puisque l'on travaille sur des systèmes finis). Ainsi, si l'on découvre un état terminal qui ne respecte pas la condition selon laquelle les jetons doivent tous être dans les place `END`, on peut conclure à un *deadlock*.

Comme nous l'avons vu précédemment, dans Quasar, lorsqu'une tâche appelle une entrée, elle dépose un jeton avec son identifiant et l'identifiant de la tâche appelée dans une place `CALL`. La tâche appelée récupère alors le jeton portant son identifiant, effectue le traitement puis dépose une réponse dans la place `RETURN` permettant ainsi à la tâche appelante de continuer.
Supposons alors que la tâche `Task_Server` n'effectue une boucle que de 1 à `MAX_CLIENT`-1, la tâche appelée ne récupérera jamais le jeton du dernier appelant dans la place `CALL`. Le client sera alors bloqué et n'atteindra jamais la place `END`. Le *deadlock* sera alors détecté par Quasar.

Imaginons alors un second cas plus subtil dans lequel la variable `The_Task_Server` est cette fois-ci un pointeur. Supposons alors que ce pointeur soit alloué dans la procédure principale. Lorsqu'un client appelle l'entrée de cette tâche allouée, deux cas sont alors possibles :

1. le client effectue son appel à l'entrée de `The_Task_Server` *avant* que la tâche appelée ait été allouée ; dans ce cas, la tâche appelante n'aura pas le bon identifiant de tâche à appelée et sera bloquée indéfiniment ce qui aboutit à un *deadlock*.
2. le client effectue son appel à l'entrée de `The_Task_Server` *après* que la tâche appelée ait été allouée ; dans ce cas le programme s'exécute correctement.

Le comportement de ce programme modifié dépend de l'ordre dans lequel les allocations sont effectuées. Ce type d'erreur n'est pas forcément détecté par le compilateur et n'apparaît pas à chaque exécution alors que Quasar, en explorant toutes les exécutions possibles, détectera forcément ce type d'erreur puisqu'il modélise exactement les mécanismes internes du langage Ada.

7.5 Conclusion & perspectives

Dans ce chapitre, nous avons présenté la plate-forme Quasar pour la vérification automatique de programmes Ada concurrents. Nous avons vu qu'en utilisant les réseaux de Petri colorés comme modèle formel il était possible de modéliser des programmes Ada complexes et dynamiques.

L'une des particularité de Quasar concerne les réductions successives appliquées à chaque étape de l'analyse pour essayer de répondre au problème de l'explosion combinatoire : sur le programme initial *via* une première analyse statique du code lors du *slicing*, puis sur le réseau de Petri généré avant d'effectuer les réductions lors de la phase de model checking comme nous l'avons vu tout au cours de cette thèse.

Quasar n'est pas seulement un outil permettant la validation de programme concurrent. Nous pensons que plusieurs métriques rapportées par Quasar – notamment concernant les réductions – peuvent aider à l'analyse qualitative du code [101, 102]. Le langage Ada possédant une sémantique forte et des constructions évoluées, nous pensons qu'il peut être intéressant de l'utiliser tel un langage de spécification qu'il sera alors possible d'analyser et de valider à l'aide de Quasar.

```
procedure Dynamic_Eratho is

   task type Prime is
      entry Test_Primarity(Number : in Integer);
   end Prime;

   type Access_Prime is access Prime;

   procedure Create_Prime(New_Prime : out Access_Prime) is
   begin
      New_Prime := new Prime;
   end Create_Prime;

   task body Prime is
      My_Number    : Integer := 0;
      Temp         : Integer := 0;
      My_Next      : Access_Prime := null;
      Termination : Boolean := False;
   begin
      while not(Termination) loop
         accept Test_Primarity(Number : in Integer) do
            Temp := Number;
         end Test_Primarity;
         if (Temp = 0) then
            -- Termination message
            Termination := True;
            if (My_Next /= null) then
               -- Propagate the termination message
               My_Next.Test_Primarity(0);
            end if;
         else
            if (My_Number = 0) then
               -- Store the prime number
               My_Number := Temp;
            else
               -- Test the primarity
               if ((Temp mod My_Number) /= 0) then
                  if (My_Next = null) then
                     -- If no neighbor, create one
                     Create_Prime(My_Next);
                  end if;
                  -- Test the primarity on the neighbor
                  My_Next.Test_Primarity(Temp);
               end if;
            end if;
         end if;
      end loop;
   end Prime;

   My_Prime : Prime;

begin
   for Number in 2..5 loop
      My_Prime.Test_Primarity(Number);
   end loop;
   -- Send the termination message
   My_Prime.Test_Primarity(0);
end Dynamic_Eratho;
```

FIG. 7.22 – Le crible d'Eratosthene

Chapitre

8

Cyclades

Sommaire

Cyclades est un model checker parallèle et réparti développé dans le cadre de cette thèse. Comme nous l'avons vu dans le chapitre précédent, Cyclades est une brique logicielle de la plate-forme Quasar. Cependant, Cyclades est un outil totalement autonome qui peut être utilisé en dehors du contexte de Quasar.

Actuellement Cyclades permet la vérification à la volée de propriétés d'accessibilité sur des réseaux de Petri colorés. Il est basé sur le model checker Helena que nous présenterons dans une première partie. Nous présenterons rapidement ses principales caractéristiques puis nous étudierons celles de Cyclades.

La seconde partie sera consacrée à une fonctionnalité particulière de Cyclades : le calcul de flots colorés. Nous présenterons alors quelques définitions – dont notamment celle des réseaux de Petri bien-formés simples sur lesquels nous calculons ces flots colorés – avant de présenter l'algorithme permettant le calcul de ces invariants.

8.1 Quelques caractéristiques de Cyclades

8.1.1 Le model checker Helena

a. Langage de spécification Helena

Afin de pouvoir vérifier efficacement des réseaux fournis par Quasar, une attention particulière a été portée au pouvoir d'expressivité fourni : le formalisme d'Helena doit être d'un niveau suffisamment

élevé pour pouvoir aisément traduire des programmes Ada concurrents en réseaux colorés. La possibilité de pouvoir définir des types de données évolués, p.ex., des types structurés, ainsi que des fonctions complexes écrites dans une syntaxe proche de celle du langage Ada permet dans une large mesure de répondre à ce besoin.
Helena fournit à l'utilisateur la possibilité de définir des types de données évolués. Ces types viennent se ranger dans quatre catégories : les types numériques, les types énumérés, les types structurés, et les types vecteurs (ou tableaux). Les domaines de couleur des places sont ensuite construits à partir de ces types de données. Plus exactement, ce sont des produits cartésiens de types de données. Il devient ainsi aisé de traduire les types de données des langages de programmation en types Helena. Quasar procède d'ailleurs de la manière suivante : chaque type du programme est associé à un type Délaina. En outre, la possibilité de pouvoir définir des types vecteurs à plusieurs dimensions ou indicés par des valeurs énumérés facilite la traduction des types array du langage Ada.

Une autre possibilité intéressante de l'outil est de pouvoir définir des fonctions complexes. Ces fonctions sont écrites dans un langage impératif dont la syntaxe est proche de celle du langage Ada. Ce langage contient toutes les instructions des langages de programmation évolués : boucles, instructions conditionnelles, ... Une fois définies, ces fonctions peuvent ensuite apparaître dans les fonctions de couleur qui étiquettent les arcs du réseau.
Cette fonctionnalité peut aussi être utile lors de la traduction d'un programme vers un réseau coloré. Toute séquence d'instructions qui n'inclut pas de synchronisation, p.ex., qui n'accède qu'à des variables locales, peut alors être traduite en une fonction Helena, pourvu que cette séquence puisse être exprimée dans notre langage.

Exemple Un exemple de réseau de Petri coloré spécifié dans le langage Helena est présenté à la figure 8.1. La première ligne correspond à une déclaration de constante. La syntaxe est largement inspirée du langage Ada. Les deux lignes suivantes correspondent à la déclaration de types. Ces types sont alors de nouveaux domaines de couleur. Suivent deux déclaration de fonction (`least` et `incr`), là encore inspirées de la syntaxe Ada.
La spécification d'une place est assez similaire à la syntaxe utilisée par l'outil Prod. Considérons la place `lbIdle`, la première partie (`dom`) correspond au domaine de couleur de la place. La seconde partie (`init`) spécifie le marquage initial de la place.
La spécification de la transition `lbReceiveRequest` comprend deux partie : la première (`in`) correspond aux entrées de la transitions – à savoir deux jetons pris dans les places `lbIdle` et `clientsRequest` – tandis que la seconde (`out`) correspond aux jetons produits par la transition – à savoir deux jetons dans les places `lbIdle` et `serversRequest`. Il est également possible de définir des gardes (cf. transition `lbBalance`).

b. Compilation et exécution du modèle

Les trois opérations suivantes représentent le principal goulot d'étranglement des performances d'un model checker explicite :
- Le test de franchissabilité : déterminer les transitions franchissables en un état
- L'exécution d'une transition : calculer le successeur d'un état
- Le test d'appartenance à l'espace d'état : déterminer si un état a déjà été visité et l'insérer dans l'espace d'état si il n'y est pas

La complexité des deux premières opérations dépend en grande partie du formalisme du model checker. Ainsi, pour le model checker Spin basé sur le langage de description de protocoles Promela, ces opérations sont négligeables et c'est principalement la troisième opération qui représentera une large fraction du temps total d'exécution. Dans le cadre des réseaux de Petri colorés, les deux premières opérations sont elles aussi très coûteuses.

Une technique reconnue pour accélérer ces trois opérations, et plus particulièrement les deux premières, consiste à compiler le modèle en un exécutable qui correspond au "vrai" model checker.

```
constant int N := 5;

type Cs : range 0 .. N;
type L : vector [Sid] of Cs;

// retourne l'indice du serveur le moins charge dans l
function least(L l) -> Sid {
   Sid result := Sid'first;
   for(s in Sid) if(l[s] < l[result]) result := s;
   return result;
}
// incremente la charge du serveur s dans le vecteur l
function incr(L l, Sid s) -> L  { return l :: ([s] := l[s] + 1); }

place lbIdle    { dom : L; init : <([0])>; }

transition lbReceiveRequest {
   in  { lbIdle         : <(l)>;
         clientsRequest : <(c)>; }
   out { lbIdle         : <(incr(l, least(l)))>;
         serversRequest : <(c, least(l))>; }}

// retourne l'indice du serveur le plus charge dans l
function most(L l) -> Sid {
   Sid result := Sid'first;
   for(s in Sid) if(l[s] > l[result]) result := s;
   return result;
}

// decremente la charge du serveur s dans le vecteur l
function decr(L l, Sid s) -> L  { return l :: ([s] := l[s] - 1); }

// teste si le vecteur de charges l est bien balance, i.e., pour tout
// couple de serveur (s1, s2), on a : |l[s1] - l[s2]| <= 1
function balanced(L l) -> bool {
   Cs smax := 0; Cs smin := Cs'last;
   for(i in Sid) { if(l[i] > smax) smax := l[i];
                   if(l[i] < smin) smin := l[i]; }
   return (smax - smin) <= 1;
}

place lbBalancing { dom : L; }

transition lbReceiveNotification {
   in  { lbIdle                : <(l)>;
         serversNotification   : <(s)>; }
   out { serversNotificationAck : <(s)>;
         lbBalancing           : <(decr(l, s))>; }}

transition lbBalance {
   in  { lbBalancing    : <(l)>;
         serversRequest : <(c, most(l))>; }
   out { lbIdle         : <(decr(incr(l, least(l)), most(l)))>;
         serversRequest : <(c, least(l))>; }
   guard : not balanced(l);}

transition lbNoBalance {
   in  { lbBalancing : <(l)>; }
   out { lbIdle      : <(l)>; }
   guard : balanced(l);}
```

FIG. 8.1 – Extrait de la spécification du répartiteur de charge

FIG. 8.2 – Le processus de compilation du réseau

Celui-ci est en quelque sorte un model checker dédié. Procéder d'une telle manière présente un avantage considérable : les expressions qui apparaissent dans les fonctions de couleurs sont directement exécutées au lieu d'être évaluées par un interpréteur. Ainsi, l'accélération obtenue dépend directement de la complexité des fonctions de couleur du réseau.

Cette technique de compilation du modèle est aussi implémentée dans les outils Spin, Prod, et Maria. Les travaux de Mäkelä ont montré l'intérêt de compiler le modèle au lieu de l'interpréter ([140], Table 1) : même sur un modèle simple (la base de données distribuée) le temps d'exécution a été divisé par 5. De plus, la compilation du fichier source généré est un désavantage négligeable par rapport au gain que nous pouvons espérer.
Helena compile aussi les réseaux colorés en exécutables. Pour des raisons de performance et de portabilité, le langage cible utilisé est le C. Afin de faciliter la lecture et la correction des bugs, le code généré est commenté, formatté, et structuré en plusieurs librairies.
Le schéma de la figure 8.2 illustre ce processus de compilation du réseau. Helena prend en entrée un fichier "lna" qui est la spécification du réseau. Cette spécification est compilée par Helena en un fichier C qui est lui-même compilé en un exécutable qui est lancé.

8.1.2 Le model checker Cyclades

Le model checker Cyclades est la version parallélisée de l'outil Helena. Il utilise le même langage de spécification de réseaux de Petri colorés et offre donc le même pouvoir d'expression. Ceci permet notamment d'utiliser l'un ou l'autre des model checker pour la vérification d'un programme traduit en réseau de Petri coloré sans avoir à effectuer aucune modification.

Cyclades se base également sur le même procédé de compilation du réseau pour générer un programme intermédiaire écrit un C. Il réutilise alors une grande partie des librairies d'Helena.

Du point de vue de l'implémentation, Cyclades rajoute une couche de communication supplémentaire à Helena. Cette couche de communication est basée sur la librairie OpenMPI[1] pour la gestion des communications. Cyclades redéfinit également les procédures d'exploration de l'espace d'état pour prendre en compte l'aspect réparti.

Pour fonctionner, Cyclades se base sur le partage du système de fichier (*via* NFS). Ceci permet de ne pas avoir à effectuer de déploiement sur les machines utilisées pour la vérification.

La version multithreadée de Cyclades requiert également l'utilisation de *threads* que nous avons géré en utilisant les *threads* POSIX. Là encore l'aspect multithreadé a nécessité de modifier légèrement les méthodes d'exploration de l'espace d'état mais également de rajouter des mécanismes de verrou pour protéger les données partagées telles que l'espace d'état ou les tables de compression pour la représentation efficace des états.

8.2 Calcul de semi-flots colorés

Les semi-flots permettent d'obtenir de nombreuses informations comportementales qui peuvent être utilisées dans l'optique d'améliorer l'analyse d'un modèle. Bien que le calcul (d'une famille

[1] `www.open-mpi.org/`

génératrice) de semi-flots dans les réseaux de Petri ordinaires soit un problème bien maîtrisé, le calcul d'une famille génératrice de semi-flots dans les réseaux de Petri colorés reste un problème ouvert. Nous présentons dans ce chapitre, une approche pragmatique nous permettant de définir un algorithme pour le calcul de famille génératrice de semi-flots pour un grand sous-ensemble de réseaux de Petri colorés : les réseaux de Petri bien formés simples.

Parmi les techniques d'analyse structurelle, le calcul d'invariant peut être considéré comme l'une des techniques fondamentales : les invariants permettent d'obtenir des informations comportementales statiquement qui permettent notamment d'effectuer des réductions structurelles (comme la réduction de places implicites [10, 12] ou l'agglomération de transitions [17, 22]), les invariants de places ou de transitions permettent également de définir/classer différents types de réseaux de Petri avec des conditions de vivacités simplifiées [4], ... Nous verrons également dans cette thèse que ces informations comportementales peuvent nous aider dans le cadre d'algorithmes de vérification répartis. Parmi les différents types d'invariants, les flots positifs (flots qui n'utilisent que des pondérations positives) sont les plus utiles et donnent des informations précises sur le réseau.

Dans les réseaux de Petri ordinaires, les invariants sont calculés à l'aide de l'algorithme de Gauss lorsqu'il n'y a pas de contraintes positives ou avec l'algorithme de Farkas lorsque l'on veut générer une famille génératrice de flots positifs. Ces deux algorithmes sont décrits dans [11].

Cependant, dans de nombreux cas, on utilise les réseaux de Petri colorés pour modéliser les algorithmes. Ils permettent en effet une description plus simple d'un problème que les réseaux de Petri ordinaires. De plus, définir des modèles paramétrés permet d'étudier différentes solutions sur un modèle unique. Calculer des invariants colorés est alors un problème intéressant. Cependant, leur calcul soulève de nouveaux problèmes : il faut désormais manipuler des *mappings* de couleur au lieu de simples entiers et les modèles pouvant être paramétrés, les algorithmes doivent prendre en compte cette nouvelle complexité. Lorsque l'on chercher à calculer des flots, deux approches sont possibles : en généralisant l'algorithme de Gauss pour prendre en compte les *mappings* de couleur (en utilisant la notion de "generalized inverse") [20, 23, 27, 15] ; cette première approche permet d'obtenir une famille génératrice (pour le dernier article) mais nécessite de fixer les paramètres et calcule des flots qui ne sont pas toujours aisément utilisables. La seconde approche consiste à restreindre les réseaux colorés ; ceci aboutit alors à différents algorithmes qui calculent des flots intéressants et paramétrés dans les réseaux réguliers [21] ou les réseaux ordonnés et associatifs [16].

Néanmoins, un seul algorithme existe aujourd'hui pour le calcul de flots colorés positifs [13] et ne fonctionne que sur des réseaux extrêmement restrictifs : les réseaux unaires réguliers ou les réseaux prédicats/transitions unaires.
Nous présentons dans cette section un algorithme calculant des flots positifs pour un modèle de réseaux de haut-niveau : les réseaux bien formés simples. Ce modèle est une restriction des réseaux bien-formés [1], mais suffisante pour la modélisation de programmes Ada concurrents.

Cette section est organisée comme suit : après quelques définitions, nous montrerons comment tirer profit des restrictions syntaxiques utilisées pour définir les réseaux et des flots positifs pour obtenir une "fractale" particulière et un système d'équations "régulier". Nous proposerons ensuite un algorithme permettant le calcul d'une famille génératrice de flots positifs simples (notons que cette famille ne génère pas *tous* les flots positifs).

8.2.1 Définitions

a. Réseaux de Petri colorés

Comme nous l'avons vu dans le chapitre 1, les réseaux de Petri sont un formalisme extrêmement intéressant pour exprimer et analyser des comportements concurrents [9, 25]. Cependant, il est souvent difficile de modéliser des problèmes complexes du fait de l'aspect très bas niveau d'expres-

sivité fourni les réseaux de Petri. C'est pourquoi il est préférable d'avoir recours aux réseaux de Petri colorés.

Nous réutiliserons ici les nombreuses définitions présentées dans le chapitre 1 que nous compléterons avec quelques nouvelles définitions qui nous permettront de présenter notre algorithme de calcul de flots colorés.
Nous noterons $\epsilon = \{\bullet\}$ le domaine de couleur réduit à l'unique valeur $\bullet$ (le jeton) ; ceci nous permet de considérer les réseaux de Petri ordinaires comme des réseaux de Petri colorés particuliers (le domaine de couleur étant ϵ).

Considérons alors le réseau de Petri coloré coloré présenté à la figure 8.3. Ce réseau modélise une solution au problème du dîner des philosophes (avec $N-1$ chaises).

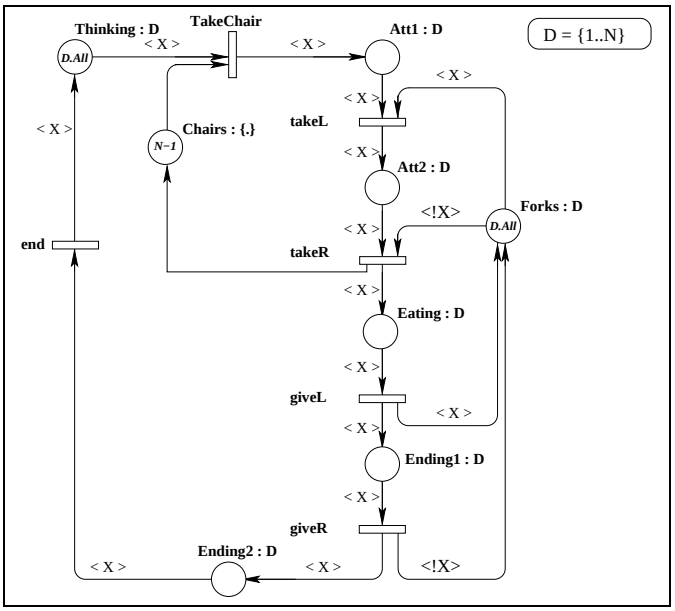


FIG. 8.3 – Un réseau coloré pour le dîner des philosophes.

Dans ce modèle, `X` représente le *mapping* "identité" vers un ensemble fini D et `!X` représente le *mapping* "successeur" vers D (i.e. le *mapping* qui associe à un élément x son successeur dans D qui est supposé ordonné).
La sémantique de ce modèle est assez simple : un philosophe x qui souhaite manger commence par prendre une chaise en franchissant la transition `takeChair`. Puis il doit prendre successivement sa fourchette (transition `takeL`) et celle de son voisin de droite (transition `takeR`) afin d'atteindre l'état (la place) `Eating` ; la chaise est alors rendue dès que le philosophe obtient ses deux fourchettes (transition `takeR`). Un philosophe en train de manger peut alors retourner dans l'état `Thinking` en franchissant en séquence les transitions `giveL`, `giveR` puis `end`.
Les phénomènes incorrects qui peuvent être mis en lumière par ce paradigme des philosophes sont le *deadlock* et les problèmes de famine.
De nombreuses techniques d'analyse ont été adaptées pour les réseaux colorés dont, en particulier, le calcul automatique d'invariants de places [7, 26, 23, 27, 28, 21, 15, 14]. En effet, les invariants de places donnent de précieuses informations sur le comportement du modèle sans nécessité d'"exécuter" le modèle ; leur définition et leur calcul ne requiert en effet que la structure du modèle.
Au sein de ces différents invariants de places, les flots positifs sont ceux qui fournissent les informations les plus intéressantes ; ceci s'explique notamment par le fait que les contraintes positives sur les pondérations simplifient l'interprétation de ces invariants.
Nous rappelons alors la définition de ces invariants de places.

Définition 8.1 (Flot coloré positif)**.** *Soit C_{inv} un domaine de couleur. Un flot positif $\mathcal{F}$, avec comme domaine de couleur C_{inv} ($\mathcal{C}(\mathcal{F}) = C_{inv}$), est un vecteur sur P, noté comme la somme $\mathcal{F} = \sum_{p\in P} \mathcal{F}_p.p$, tel que $\forall p \in P, \mathcal{F}_p$ est un mapping de $Bag(\mathcal{C}(p))$ vers $Bag(C_{inv})$ et tel que $\forall t \in T$, $\sum_{p\in P} \mathcal{F}_p \circ W(t,p) = 0$* [1].

Cette définition implique que pour n'importe quel flot $\mathcal{F}$, et pour tout marquage accessible m, l'égalité $\sum_{p\in P} \mathcal{F}_p(m(p)) = \sum_{p\in P} \mathcal{F}_p(m_0(p))$ est vérifiée. Un flot peut alors être interprété comme un ensemble d'équations liants le marquage d'un sous-ensemble de place au marquage initial de ces places :

$$\forall m \in Acc(CN, m_0), \forall c \in \mathcal{C}(\mathcal{F}), \sum_{p\in P} \sum_{c_p \in \mathcal{C}(p)} [\mathcal{F}_p(c_p)(c)].m(p)(c_p) = cst(\in \mathbb{N})$$

Considérons de nouveau le modèle de la figure 8.3. Sur ce réseau, nous pouvons identifier au moins trois flots positifs sur le domaine de couleur D :
– $\mathcal{F}_1 = \langle X\rangle.Thinking + \langle X\rangle.Att1 + \langle X\rangle.Att2 + \langle X\rangle.Eating + \langle X\rangle.Ending1 + \langle X\rangle.Ending2$
– $\mathcal{F}_2 = \langle !X\rangle.Forks + \langle !X\rangle.Att2 + \langle !X\rangle.Eating + \langle X\rangle.Eating + \langle X\rangle.Ending1$
– $\mathcal{F}_3 = \langle .\rangle.Att1 + \langle .\rangle.Att2 + \langle 1\rangle Chairs$
où $\langle X\rangle$ représente le *mapping* "identité" sur $Bag(D)$, $\langle !X\rangle$ le *mapping* "successeur" sur $Bag(D)$, $\langle 1\rangle$ le *mapping* "identité"sur ϵ, et $\langle .\rangle$ la projection de D vers *epsilon* définie par $\forall d \in D$, $.(d) = \bullet$. Notons que par soucis de simplicité nous omettons la notation $\langle 1\rangle$.
Le premier invariant caractérise la structure séquentielle des philosophes qui peuvent être dans l'un des six états `Thinking`, `Att1`, `Att2`, `Eating`, `Ending1`, `Ending2`. En effet, son interprétation est [2] : $\forall m \in Acc(CN, m_0), \forall x \in D, m(Thinking)(x) + m(Att1)(x) + m(Att2)(x) + m(Eating)(x) + m(Ending1)(x) + m(Ending2)(x) = 1$.

Le second invariant, $\mathcal{F}_2$, spécifie que, étant donné $x \in D$, la fourchette$!x$ est soit libre (place `Forks` marquée par $!x$), soit détenue par le philosophe $!x$ qui est dans l'état `Eating` ou `Att2` , soit utilisée par le philosophe x qui est dans l'état `Eating` ou `Ending`.
Enfin, le dernier invariant $\mathcal{F}_3$ met en avant le fait que les chaises sont soit libres ((place `Chairs` marquée), soit partagées par des philosophes qui sont dans les états `Att1` ou `Att2`. Son interprétation est la suivante : $\forall m \in Acc(CN, m_0), \sum_{x\in C} m(Att1)(x) + \sum_{x\in C} m(Att2)(x) + m(Chairs) = N - 1$ et, combinée avec $\mathcal{F}_2$, garantit que la place `Forks` ne peut devenir vide et donc qu'aucun *deadlock* n'est possible.

Nous voyons alors ici tout l'intérêt des flots colorés qui nous permettent d'obtenir des informations comportementales en utilisant uniquement la structure du réseau.
Cependant, le calcul des flots positifs reste une tâche difficile. Une explication possible réside dans le fait que la plupart des recherches se sont concentrées sur le calcul de familles génératrices (i.e. de familles à partir desquelles il est possible de générer tous les invariants positifs du réseau). Ceci aboutit à des systèmes d'équations extrêmement complexes même si les algorithmes sont définis à partir de réseaux colorés fortement contraints. Nous ne proposons pas ici de nous concentrer sur les familles génératrices mais uniquement sur un sous-ensemble utile de familles de flots positifs. Pour cela, nous imposons une restriction syntaxique sur la définition des flots positifs et associons cette restriction à une restriction similaire sur la définition des réseaux colorés. Nous appelons alors ce type de réseau les "réseaux bien-formés simples" et notons alors nos flots positifs comme "flots positifs simples".

b. Réseaux colorés bien-formés simples et flots positifs simples

La modélisation et la vérification sont deux concepts fortement liés. Un formalisme doit définir un bon compromis entre la simplicité de modélisation et la richesses des éventuels outils automatiques et des techniques permettant la vérification de propriétés sur ce type de modèle.

[1] 0 représente ici le *mapping* nul de $\mathcal{C}(t)$ vers $Bag(\mathcal{C}(\mathcal{F}))$
[2] Rappelons que $\langle X\rangle(c)(c') = 1$ si $c = c'$ et $\langle X\rangle(c)(c') = 0$ sinon.

Dans le domaine des réseaux de Petri, des formalismes généraux ont été proposés (comme les réseaux colorés) mais la plupart des résultats théoriques ont été obtenus en utilisant des restrictions sur les modèles généraux. Citons par exemple les réseaux réguliers [21], les réseaux ordonnés [16], ainsi qu'un formalisme avec les mêmes possibilités d'expression que les réseaux colorés mais avec quelques restrictions syntaxique : les réseaux bien-formés [1].
Nous proposons ici une légère restriction de ce dernier formalisme que nous appellerons les réseaux bien-formés simple (ou réseaux SWF). Comme nous l'avons précisé précédemment, ce formalisme reste suffisant pour modéliser des problèmes complexes dont un grand nombre de programmes Ada qui peuvent ensuite être validés.
Nous définissons alors ce nouveau formalisme *via* certaines restrictions sur les *mappings* de couleur ainsi que sur la construction des domaines de couleur.

Définition 8.2 (Mapping coloré basique)**.** *Soit C un ensemble fini ordonné. Les mappings basiques sont l'* identité*, notée X_C, la* diffusion *(également appelée synchronisation globale), notée All_C, le* successeur*, noté $!X_C$ et tous les mappings* constants*, $\lambda_C^c, c \in C$. Ils sont définis de C vers $Bag(C)$ par : $\forall x \in C$, $X_C(x) = x$, $All_C(x) = \sum_{c \in C} c$, $!X_C(x) =$ successor of x in C et $\lambda_C^c(x) = c$.*

Remarque. On utilise communément des lettres $X, Y, Z, Ph, \ldots$ pour noter les *mappings* identité ou successeur et, lorsque le contexte le permet, on omet régulièrement le domaine sur lequel s'appliquent ces *mappings* (X au lieu de X_C). Toutes les classes sont considérées comme ordonnées (de façon circulaire). Ainsi, chaque élément possède un unique successeur dans la classe (le successeur du dernier élément étant alors le premier élément). Tous ces *mappings* peuvent être étendus au *mapping* de $Bag(C)$ vers $Bag(C)$. Lorsque $C = \epsilon$ tous ces *mappings* coïncident.

Définition 8.3 (Mapping coloré simple)**.** *Soit C un ensemble ordonné fini. Un mapping de couleur simple sur C est un mapping de C vers $Bag(C)$ si c'est un mapping constant ou s'il peut s'écrire comme une composition $\alpha.X_C + \beta.All_C + \gamma.!X_C$ où α, β, γ sont des entiers tels que $\beta \geq 0$, $\beta + \alpha \geq 0$ et $\beta + \gamma \geq 0$ (et $\beta + \alpha + \gamma \geq 0$ dans le cas où $|C| = 1$).*

Remarque. Les contraintes sur α, β et γ assurent qu'un *mapping* coloré défini une valeur positive pour chaque couleur de C (et appartient ainsi à $Bag(C)$). Lorsque $C = \epsilon$ tous les *mappings* colorés simple sont réduits à un *mapping* constant (une valeur entière).

Définition 8.4 (Fonction de couleur simple)**.** *Soit $C = C_1 \times C_2 \times \ldots \times C_k$ un produit fini d'ensembles finis non vides. Un mapping f de C vers $Bag(C)$ est une* fonction de couleur simple *s'il peut s'écrire $f = \langle f_1, f_2, \ldots, f_k \rangle$ où $\forall i$, f_i est un mapping coloré simple sur C_i ou une application unitaire arbitraire [1] de C_i vers $Bag(C_i)$. Si $\langle c_1, \ldots, c_k \rangle \in C$ alors $f(c) = \langle f_1, f_2, \ldots, f_k \rangle (\langle c_1, \ldots, c_k \rangle) = \langle f_1(c_1), f_2(c_2), \ldots, f_k(c_k) \rangle$.*
Lorsque cela sera nécessaire, nous noterons $\langle f_1, f_2, \ldots, f_k \rangle$ par $f_1.\langle f_2, \ldots, f_k \rangle$ et étendrons la linéarité de cette notation aux sommes pondérées de tuples.

Nous pouvons alors définir les réseaux bien-formés simples.

Définition 8.5 (Réseaux bien-formés simples)**.** *Un réseau coloré $\langle P, T, \mathcal{C}, G, W^-, W^+ \rangle$ est un* **réseau bien-formé simple** *si $\forall p \in P$, $\forall t \in T$,*
- *$W^+(t,p) \neq 0$ or $W^-(t,p) \neq 0$ implique que $\mathcal{C}(t) = \mathcal{C}(p) \times C'_p$ ($\mathcal{C}(t)$ égal ou inclus $\mathcal{C}(p)$) ;*
- *$W^+(t,p)$ et $W^-(t,p)$ sont une composition d'une fonction de couleur simple sur $\mathcal{C}(p)$ avec une projection de $\mathcal{C}(t)$ vers $\mathcal{C}(p)$ (où le domaine de couleur de t est "plus grand" que le domaine de couleur de p) ;*
- *si $W^+(t,p)$ utilise un mapping constant sur la classe C_i alors un arc entre p et une autre transition ne peut utiliser le mapping All_{C_i}.*

On dit que le réseau est **homogène** *si tous les domaines de couleurs sont identiques(i.e. $\exists C_1, \ldots, C_K$ tel que $\forall s \in P \cup T$, $\mathcal{C}(s) = C_1 \times \ldots \times C_k$), si toutes les gardes sont toujours évaluées à vrai (il n'y a pas de gardes) et si tous les mappings colorés sont uniquement construits avec les mappings basiques X, All $!X$ (les mappings constants et arbitraires ne peuvent apparaître sur les arcs).*

[1] f_i est une application unitaire si $\forall c \in C_i, |f_i(c)| = 1$

Remarque. Ce troisième point est utilisé pour assurer que l'on pourra toujours homogénéiser un réseau bien-formé simple : i.e. construire un modèle équivalent (ou qui effectue une simulation faible de l'original) mais avec un domaine de couleur unique pour chaque place et transition du réseau (cf. d.).
Les réseaux SWF sont donc une restriction des réseaux bien formés (WF). En particulier, ils n'autorisent pas les fonctions gardées ou la composition par addition de différentes instances d'une même classe de couleur (e.g. $\langle X\rangle + \langle Y\rangle$ est interdit).
Cependant, l'expressivité offerte par cette définition reste suffisante pour modéliser de nombreux problèmes et notamment un grand nombre de programmes Ada concurrents.
Nous donnons maintenant la définition des flots positifs que nous calculerons sur les réseaux SWF.

Définition 8.6 (Flots positifs simples)**.** *Un flot positif $\mathcal{F}$ sur le domaine de couleur D est qualifié de simple si $\forall p \in P$, $\mathcal{C}(p) = D \times \mathcal{C}'_p$ ($\mathcal{C}(p)$ "inclu" D), $\mathcal{F}_p$ est une composition d'un fonction de couleur simple sur $\mathcal{C}(p)$ n'utilisant pas de constante avec une projection de D vers $\mathcal{C}(p)$.*

Les restrictions apportées ne sont donc pas très fortes : en effet, les trois flots présentés précédemment pour le modèle de la figure 8.3 sont tous des flots positifs.
De plus, cette définition offre deux avantages :

1. leur définition n'utilise que des *mappings* colorés simples. Ces flots positifs sont alors aisément interpretables et utilisables (notamment dans le cadre de réductions structurelles) : ils peuvent identifier des sections critiques (comme $\mathcal{F}3$ dans notre exemple), ils peuvent également mettre en avant la structure d'un processus ($\mathcal{F}1$) ce qui peut être très intéressant dans le cadre du partitionnement de l'espace d'état ou encore spécifier la façon dont les ressources sont partagées ($\mathcal{F}2$) ;
2. ils peuvent être calculés automatiquement comme nous le verrons un peu plus loin.

Pour plus de clarté dans la notation des flots positifs, nous utiliserons le point . pour mettre en évidence la projection de $\mathcal{C}(p)$ vers un domaine de couleur du flot. Par exemple, si le domaine de couleur du flot est $D = C_1 \times C_3$ et le domaine de couleur de la place utilisée dans le flot est $C = \mathcal{C}_1 \times C_2 \times C_1 \times C_3$, nous noterons $\langle ., ., X', Z\rangle$ la composition du *mapping* $\langle X, Y, X', Z\rangle$ avec la projection Π de C vers D définie par $\Pi(x, y, x', z) = (x', z)$.

Soit un réseaux SWF homogène, on peut construire, à partir de sa matrice d'incidence W, la matrice entière $W^{n_1,\ldots,n_k}$, indexée par $(T \times C1 \times \ldots \times C_k) \times (P \times C1 \times \ldots \times C_k)$ et définie par :

$$W^{n_1,\ldots,n_k}(t, c'_1, \ldots, c'_k)(p, c_1, \ldots, c_k) = W(t,p)(\langle c'_1, \ldots, c'_k\rangle)(\langle c1, \ldots, c_k\rangle)$$

Cette construction consiste uniquement à "déplier" le *mapping* coloré qui constitue les coefficients de la matrice W. De la même façon, il est possible de "déplier" un flot positif $\mathcal{F}$ en construisant l'ensemble de vecteurs entiers définis par toutes les interprétations possibles des flots colorés. En effet, soit un flot positif $\mathcal{F}$ et une interprétation c_{inv} on peut définir le vecteur entier $\overrightarrow{F}_{c_{inv}}$ indexé par $(C1 \times \ldots \times C_k \times P)$ et défini par :

$$\overrightarrow{F}_{c_{inv}}(c_1, \ldots, c_k, p) = \mathcal{F}_p(\langle c1, \ldots, c_k\rangle)(c_{inv})$$

En utilisant ces notations, nous avons, par définition :

Proposition 8.1. *$\mathcal{F}$ est un flot positif si et seulement si $\forall c_{inv} \in C$, $W^{n_1,\ldots,n_k}.\overrightarrow{F}_{c_{inv}} = 0$.*

Ainsi, calculer des flots colorés d'un réseau coloré homogène consiste à résoudre le système $W^{n_1,\ldots,n_k}.\overrightarrow{F}_{c_{inv}} = 0$ avec, par exemple, l'algorithme de Farkas décrit dans [11]. Cependant, ceci présente deux inconvénients :

1. ce calcul nécessite de fixer les paramètres ce qui est équivalent à un dépliage du réseau et de calculer les flots positifs sur le réseau déplié ce qui est très inefficace ;
2. comme les flots calculés sont des vecteurs entiers, il est très difficile de les "recolorer" dans le cas général. Ceci aboutit alors à des invariants difficilement exploitables.

Nous allons prouver qu'il est possible de résoudre ce systèmes de façon **paramétrée** sans déplier le réseau. Pour cela, nous allons, dans un premier temps, prouver que la matrice d'incidence peut être réordonnée dans une forme "fractale".

8.2.2 Calculs de flots positifs simple pour les réseaux

Dans cette section nous allons montrer comment calculer une famille génératrice de flots positifs simples d'un réseau SWF. Notons que l'ensemble calculé génère tous les flots positifs simples mais **pas tous** les flots positifs.
Tout d'abord, nous supposons que chaque réseau considéré est homogène (nous présenterons plus loin un mécanisme permet de transformer un réseau SWF en un réseau homogène). Nous allons montrer alors que les contraintes sur les *mappings* des réseaux SWF et les définitions des flots positifs simples aboutissent à un système de forme "fractale" qui peut être réduit à un ensemble d'équations non paramétrées.
Nous utilisons alors les notations suivantes :
- $C = C_1 \times C_2 \times \ldots \times C_k$ représente le domaine de couleur commun aux places et transitions ;
- $n_1 = |C_1|$, $n_2 = |C_2|$, ..., $n_k = |C_k|$,
- $C_j = \left\{c_j^i\right\}_{i=1..n_j}$;
- $\forall p \in P, \forall t \in T$, $W(t,p) = \langle w_1(t,p), \ldots, w_k(t,p)\rangle$ avec pour tout i dans $[1..k]$, $w_i(t,p) = a_i(t,p).X_{C_i} + b_i(t,p).All_{C_i} + d_i(t,p).!X_{C_i}$
- nous calculons le flot positif sur le domaine C ; nous fixons donc $C_{inv} = C$.
- si $\mathcal{F} = \sum_p \mathcal{F}_p.p$ est un flot positif simple, on note $\mathcal{F}_p = \langle f_1^p, \ldots, f_k^p\rangle$ et $f_i^p = \alpha_i.X_i + \beta_i.All_i + \gamma_i.(!X_i)$;
- on note $E = \left(\mathbb{Q}^+\right)^P$;

a. Réordonner les équations

La matrice d'un réseaux SWF homogène peut être définie par le biais d'une construction récursive mettant en avant la forme "fractale" qui peut être utilisée pour définir un algorithme efficace pour le calcul de flots positifs simples. Dans cette construction, les transitions sont organisées en lignes (correspondants aux équations du système) et les places en colonnes (correspondants aux variables du système). Cette construction est basée sur la notion de bloc matrice que nous rappelons.

Définition 8.7 (Matrice en bloc carrée)**.** *Une matrice $A = (a_{i,j})$ dans $\mathbb{N}^{K.n \times K.m}$ est un matrice en bloc carrée d'entier si chaque $a_{i,j}$ est une matrice d'entier $m \times n$ ou une matrice en bloc carrée (auquel cas, $n = k'.n'$ et $m = k'.m'$). Nous notons $A(i,j) = a_{i,j}$ l'élément à l'intersection de la $i^{ème}$ ligne et de la $j^{ème}$ colonne.*

Définition 8.8 (Matrice de forme fractale)**.** *Une matrice en bloc $W = (w_{i,j})$ dans $\mathbb{N}^{K.n \times K.m}$ possède une "forme fractale pour les réseaux bien-formés simples" (ou forme fractale) s'il existe trois matrices A, B, D de même dimensions telles que :*

1. *les matrices A, B, D sont soit trois matrices entières soit trois matrices de forme fractale ;*
2. *les éléments de W satisfont :*

$$\forall i,j \in 1..K, w_{i,j} = \begin{cases} A+B & \text{if } i=j \\ D+B & \text{if } j=i+1 \text{ modulo } n \\ B & \text{in other cases} \end{cases}$$

Par exemple, si A, B, D sont trois matrices entières de même dimensions alors la matrice suivante de dimension $n \times n$ est de forme fractale.

$$W = \begin{bmatrix} (A+B) & (D+B) & B & \ldots & B \\ B & (A+B) & (D+B) & \ldots & B \\ \ldots & \ldots & \ddots & \ddots & \ldots \\ B & \ldots & B & (A+B) & (D+B) \\ (D+B) & B & \ldots & B & (A+B) \end{bmatrix}$$

Considérons maintenant un réseau SWF homogène et W sa matrice d'incidence. Rappelons que k représente le nombre de classes du réseau et par là même, le nombre de paramètres différents du système et que $W(t,p) = \langle w_1(t,p), \ldots, w_k(t,p)\rangle$ avec pour tout i dans $[1..k]$, $w_i(t,p) = a_i(t,p).X_{C_i} + b_i(t,p).All_{C_i} + d_i(t,p).!X_{C_i}$.

Définition 8.9 (Extraction d'une forme fractale à partir d'un réseau SWF homogène)**.** *Soient $v \in [1..k+1]$, et trois ensembles $I_A, I_B, I_D \subseteq \{n_1, n_2, \ldots, n_k\}$ nous définissons la matrice entière ou en bloc $W_v^{I_A,I_B,I_D}$ récursivement par :*

– *si $v = k+1$ alors $W_{k+1}^{I_A,I_B,I_D} = (w_{t,p})$ est la matrice entière $T \times P$*

$$w_{t,p} = \prod_{i\in I_A} a_i(t,p). \prod_{j\in I_D} d_j(t,p). \prod_{l\in I_B} b_l(t,p) \quad [1]$$

– *si $v \leq k$, alors $W_v^{I_A,I_B,I_D} = (w_{i,j})$ est la matrice en bloc carrée $n_v \times n_v$ définie par*

$$w_{i,j} = \begin{cases} W_{v+1}^{I_A\cup\{v\},I_B,I_D} + W_{v+1}^{I_A,I_B\cup\{v\},I_D} & \text{if } i=j \\ W_{v+1}^{I_A,I_B,I_D\cup\{v\}} + W_{v+1}^{I_A,I_B\cup\{v\},I_D} & \text{if } j=i+1 \text{ modulo } n_v \\ W_{v+1}^{I_A,I_B\cup\{v\},I_D} & \text{in other cases} \end{cases}$$

Nous notons W' la matrice en bloc carrée $W_1^{\emptyset,\emptyset,\emptyset}$ et lorsqu'il n'y a pas d'ambiguïtés, nous notons $A = W_2^{\{1\},\emptyset,\emptyset}$, $B = W_2^{\emptyset,\{1\},\emptyset}$ et $D = W_2^{\emptyset,\emptyset,\{1\}}$.

Proposition 8.2. *Nous avons alors les résultats suivants :*

1. *Les deux matrices W et W' sont équivalentes pour la définition d'un réseau SWF homogène ; i.e. $\forall p \in P, \forall t \in T$, $\forall c = \langle c_1, \ldots, c_k\rangle \in C$, $\forall c' = \langle c'_1, \ldots, c'_k\rangle \in C$, nous avons $W'(c_1, c'1)(c_2, c'2)\ldots(c_k, c'k)(t,p) = W(t,p)(c')(c)$.*
2. *La matrice W' est de forme fractale (si $k \geq 1$).*

Démonstration. Le point 2 est une conséquence directe de la définition de $W_1^{\emptyset,\emptyset,\emptyset}$. Pour prouver le point 1, il est suffisant de noter que $W(t,p)(\langle c_1, \ldots, c_k\rangle) = \langle w_1(t,p)(c_1), \ldots, w_k(t,p)(c_k)\rangle$ et que $w_i(t,p)(c_i)(c'_i) = b_i(t,p) + \delta_{c_i,c'_i}.a_i(t,p) + \delta_{c_i,!c'_i}.d_i(t,p)$ où δ est le symbole de Kronecker défini par $\delta_{c_i,c'_i} = 0$ si $c_i \neq c'_i$ et $\delta_{c_i,c_i} = 1$; d'où

$$W(t,p)(\langle c_1, \ldots, c_k\rangle)(\langle c'_1, \ldots, c'_k\rangle) = \Pi_{i=1..k}(b_i(t,p) + \delta_{c_i,c'_i}.a_i(t,p) + \delta_{c_i,!c'_i}.d_i(t,p))$$

Nous remarquons alors que lorsque $c_v = c'_v$ nous avons ajouté v à l'ensemble I_A, lorsque $c_v = !c'_v$ nous avons ajouté v à l'ensemble I_D et nous avons utilisé ces ensembles pour calculer les valeurs entières des dernières matrices et nous obtenons le résultat.

□

Si l'on considère un réseau SWF homogène avec deux classes (deux paramètres), la matrices d'incidence peut être écrite :

$$W = \begin{bmatrix} (A+B) & (D+B) & B & \ldots & B \\ B & (A+B) & (D+B) & \ldots & B \\ & & \ddots & \ddots & \\ \ldots & \ldots & & & \ldots \\ B & \ldots & B & (A+B) & (D+B) \\ (D+B) & B & \ldots & B & (A+B) \end{bmatrix}$$

avec

$$A = W_2^{\{1\},\emptyset,\emptyset} = \begin{bmatrix} (AA+AB) & (AD+AB) & AB & \ldots & AB \\ AB & (AA+AB) & (AD+AB) & \ldots & AB \\ & & \ddots & \ddots & \\ \ldots & \ldots & & & \ldots \\ AB & \ldots & AB & (AA+AB) & (AD+AB) \\ (AD+AB) & AB & \ldots & AB & (AA+AB) \end{bmatrix}$$

[1] Dans ce produit, nous utilisons la convention selon laquelle un produit sur un ensemble vide vaut 1 ($\prod_{i\in\emptyset} f(i) = 1$)

$$B = W_2^{\emptyset,\{1\},\emptyset} = \begin{bmatrix} (BA+BB) & (BD+BB) & BB & \dots & BB \\ BB & (BA+BB) & (BD+BB) & \dots & BB \\ & & \ddots & \ddots & \\ \dots & \dots & & \ddots & \dots \\ BB & \dots & BB & (BA+BB) & (BD+BB) \\ (BD+BB) & BB & \dots & BB & (BA+BB) \end{bmatrix}$$

$$D = W_2^{\emptyset,\emptyset,\{1\}} = \begin{bmatrix} (DA+DB) & (DD+DB) & DB & \dots & DB \\ DB & (DA+DB) & (DD+DB) & \dots & DB \\ & & \ddots & \ddots & \\ \dots & \dots & & \ddots & \dots \\ DB & \dots & DB & (DA+DB) & (DD+DB) \\ (DD+DB) & DB & \dots & DB & (DA+DB) \end{bmatrix}$$

et avec
- $AA(t,p) = a_1(t,p).a_2(t,p)$, $AB(t,p) = a_1(t,p).b_2(t,p)$, $AD(t,p) = a_1(t,p).d_2(t,p)$
- $BA(t,p) = b_1(t,p).a_2(t,p)$, $BB(t,p) = b_1(t,p).b_2(t,p)$, $BD(t,p) = b_1(t,p).d_2(t,p)$
- $DA(t,p) = d_1(t,p).a_2(t,p)$, $DB(t,p) = d_1(t,p).b_2(t,p)$, $DD(t,p) = d_1(t,p).d_2(t,p)$

b. Réordonner les solutions et simplifier les équations

Tout d'abord, notons que chaque flot positif simple $\mathcal{F}$ peut être écrit de manière unique comme une somme $F_{<>} = \langle X_1, X_2, \dots, X_k\rangle f_1 + \langle X_1, X_2, \dots, !X_k\rangle f_2 + \langle X_1, X_2, \dots, All_k\rangle f_3 + \dots + \langle All_1, All_2, \dots, All_k\rangle f_{3^k}$ avec f_i des vecteurs entiers sur P.
Puis, remarquons que la réorganisation effectuée sur la matrice d'incidence peut également être appliquée aux solution du système étudié.
De fait, un flot positif $\mathcal{F}$ définit pour chaque valeur $c_{inv} \in C$ un vecteur développé $\overrightarrow{F}_{c_{inv}}$ dans $E^{n_1\times\dots\times n_k}$. Remarquons également que ce vecteur peut être vu comme un vecteur de $(E^{n_2\times\dots\times n_k})^{n_1}$ i.e. un vecteur de taille n_1 avec chaque composant dans $E^{n_2\times\dots\times n_k}$.

$$\overrightarrow{F}_{c_{inv}} = \begin{bmatrix} \overrightarrow{F[1]}_{c_{inv}} \\ \vdots \\ \overrightarrow{F[n_1]}_{c_{inv}} \end{bmatrix} \text{ with } \forall i \in 1..n_1, \overrightarrow{F[i]}_{c_{inv}}(c_2, \dots, c_k, p) = \overrightarrow{F}_{c_{inv}}(c_1^i, c_2, \dots, c_k, p)$$

Comme nous restreignons le calcul des flots positifs au calcul de flots positifs simples, nous pouvons utiliser cette forme pour de tels vecteurs et les écrire de façon paramétrée.

Proposition 8.3. *Soit $\mathcal{F}$ un flot positif* **simple** *et c_{inv} une interprétation de couleur, alors $\overrightarrow{F}_{c_{inv}}$ a une unique décomposition :*

$$\overrightarrow{F}_{c_{inv}} = \begin{bmatrix} 0 \\ \dots \\ 0 \\ F_X \\ 0 \\ \dots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \dots \\ 0 \\ 0 \\ F_{!X} \\ \dots \\ 0 \end{bmatrix} + \begin{bmatrix} F_{All} \\ \dots \\ F_{All} \\ F_{All} \\ F_{All} \\ \dots \\ F_{All} \end{bmatrix}$$

où F_X, $F_{!X}$ et F_{All} sont trois vecteurs $E^{n_2\times\dots\times n_k}$ dépendants de la valeur de $\{\alpha_i^p\}_p$ pour F_X, de $\{\gamma_i^p\}_p$ pour $F_{!X}$ et de $\{\alpha_i^p, \gamma_i^p, \beta_i^p\}_p$ pour F_{All}, et tels que si, $c_{inv} = \langle c_1^i, c_2, \dots, c_k\rangle$, alors F_X est sur la i^{eme} ligne et $F_{!X}$ est sur la $(i+1)^{eme}$ ligne.

Démonstration. Soit $c_{inv} = \langle c_1'^i, c_2', \dots, c_k'\rangle$ et $j, j' \in 1..n_1$ tout deux distincts de i et $i+1$. Supposons que $\overrightarrow{V}$ est le vecteur défini par $\overrightarrow{V} = \overrightarrow{F[j]}_{c_{inv}} - \overrightarrow{F[j']}_{c_{inv}}$. Nous avons $\forall p, c_2, \dots, c_k$, $\overrightarrow{V}(p, c_2, \dots, c_k) = \overrightarrow{F}_{c_{inv}}(p, c_1^j, c_2, \dots, c_k) - \overrightarrow{F}_{c_{inv}}(p, c_1^{j'}, c_2, \dots, c_k)$.
Ainsi, $\overrightarrow{V}(p, c_2, \dots, c_k) = \mathcal{F}_p(\langle c_1^j, \dots, c_k\rangle)(c_{inv}) - \mathcal{F}_p(\langle c_1^{j'}, \dots, c_k\rangle)(c_{inv})$ qui peut être écrit $\langle f_1^p(c_1^j, c_1^i), f_2^p(c_2, c'2), \dots, f_k^p$ $- \langle f_1^p(c_1^{j'}, c_1^i), f_2^p(c_2, c'2), \dots, f_k^p(c_k, c'k)\rangle$. Comme j et j' sont **tout deux** distincts de i et $i+1$ on a alors $f_1^p(c_1^j, c_1^i) - f_1^p(c_1^{j'}, c_1^i) = 0$ et donc $\overrightarrow{V} = \overrightarrow{0}$. Nous notons F_{All} le vecteur $\overrightarrow{F[j]}_{c_{inv}}$.

Soit $\overrightarrow{V_X}$ le vecteur définit par $\overrightarrow{V_X} = \overrightarrow{F[i]}_{c_{inv}} - \overrightarrow{F[j]}_{c_{inv}}$.
En utilisant la même argumentation, il vient que $\overrightarrow{V_X}(p, c_2, \ldots, c_k) = \langle f_1^p(c_1^i, c_1^i), f_2^p(c_2, c'2), \ldots, f_k^p(c_k, c'k)\rangle$ $- \langle f_1^p(c_1^j, c_1^i), f_2^p(c_2, c'2), \ldots, f_k^p(c_k, c'k)\rangle$. Comme $f_1^p(c_1^i, c_1^i) - f_1^p(c_1^j, c_1^i) = \alpha_i^p$ on peut noter $F_X = V_X$. Nous pouvons alors procéder de la même façon pour définir de façon unique $F_{!X} = V_{!X}$.

□

Maintenant, si nous combinons la forme fractale régulière d'un réseau SWF homogène avec la forme particulière de flots positifs simples, on peut simplifier le système à résoudre.

Proposition 8.4. *En utilisant les notations précédentes nous avons* $W_{n_1,\ldots,n_k}.\overrightarrow{F}_{c_{inv}} = 0$ *ssi :*

$$B.F_{All} = A.F_{!X} = D.F_X = A.F_X + D.F_{!X} = B.(F_X + F_{!X}) + (A + D).F_{All} = 0$$

Démonstration. Le système peut être écrit

$$\begin{bmatrix} (A+B) & (D+B) & B & \ldots & B \\ B & (A+B) & (D+B) & \ldots & B \\ & & \ddots & \ddots & \\ \ldots & \ldots & & & \ldots \\ B & \ldots & B & (A+B) & (D+B) \\ (D+B) & B & \ldots & B & (A+B) \end{bmatrix} . \begin{bmatrix} F_{All} \\ \ldots \\ F_{All} \\ F_X + F_{All} \\ F_{!X} + F_{All} \\ \ldots \\ F_{All} \end{bmatrix} = 0$$

Puisque les flots positifs sont définis seulement avec des vecteurs entiers (ils n'utilisent pas n_1 comme coefficient) et puisqu'un flot positif simple défini des solution pour n'importe quelle valeur de n_1, il vient que $B.F_{All} = 0$. Si l'on développe les équations (en utilisant le fait que $B.F_{All} = 0$), on obtient seulement les quatre équations :
– $(A + D).F_{All} + B.(F_X + F_{!X}) = 0$;
– $(A + B).F_X + (B + D).F_{!X} + (A + D).F_{All} = 0$;
– $B.F_X + (A + B).F_{!X} + (A + D).F_{All} = 0$;
– $(B + D).F_X + B.F_{!X} + (A + D).F_All = 0$.
En effectuant une soustraction de la première sur les autres on obtient alors le résultat. Maintenant, si F_X, F_{All} et $F_{!X}$ valident la précédente équation il est alors évident que le vecteur $\overrightarrow{F}_{c_{inv}}$ est solution de $\overrightarrow{F}_{c_{inv}} = 0$ (quelle que soit la valeur positive de n_1).

□

c. Calcul de flots positifs simple dans le cas de réseaux homogènes

Nous sommes alors en mesure de proposer un algorithme pour calculer des flots positifs simples pour un réseau SWF homogène. Pour cela, il est cependant nécessaire de définir deux opérateurs matriciels : le première, l'opérateur "empilement" définit comment empiler des matrices de même dimension. Le second, la "juxtaposition", définit comment placer des matrices de même dimensions côte à côte. Ces deux opérateurs diffèrent des opérateurs classiques dans le sens où ils conservent la structure fractale des matrices.

Définition 8.10 (Empiler et juxtaposer des matrices)**.** *Soient* $W^1 = [w_{i,j}^1]_{i\in[1..n],j\in[1..m]}$, $\ldots$, $W^q = [w_{i,j}^q]_{i\in[1..n],j\in[1..m]}$ *q des matrices.*

1. *Si* $W^1, \ldots, W^q$ *sont toutes des matrices entières, alors*
   – *l'empilement de* $W^1, \ldots, W^q$*, noté* $\left[W^1/\ldots/W^q\right]$*, est la matrice* $[s_{i,j}]_{i\in[1..q.n],j\in[1..m]}$ *(m colonnes et q.n lignes) avec* $s_{(q.i)-r,j} = w_{i,j}^{q-r}$*,* $r \in 0..q-1$*.*
   – *la juxtaposition de* $W^1, \ldots, W^q$*, notée* $\left[W^1|\ldots|W^q\right]$*, est la matrice* $[s_{i,j}]_{i\in[1..n],j\in[1..q.m]}$ *(n lignes et q.m colonnes) avec* $s_{i,(q.j)-r} = w_{i,j}^{q-r}$*,* $r \in 0..q-1$*.*
2. *Si* $W^1, \ldots, W^q$ *sont toutes des matrices en blocs (leurs éléments sont des matrices) alors*
   – *l'empilement de* $W^1, \ldots, W^q$*, noté* $\left[W^1/\ldots/W^q\right]$*, est la matrice* $[s_{i,j}]_{i,j\in[1..n]}$ *définie récursivement par* $s_{i,j} = \left[w_{i,j}^1/\ldots/w_{i,j}^q\right]$

– *la juxtaposition de* $W^1, \ldots, W^q$*, notée* $\left[W^1|\ldots|W^q\right]$*, est la matrice* $[s_{i,j}]_{i,j\in[1..n]}$ *définie récursivement par* $s_{i,j} = \left[w^1_{i,j}|\ldots|w^q_{i,j}\right]$

Remarque. Comme un vecteur peut être vu comme une matrice à une seule colonne, ces deux opérateurs peuvent s'appliquer aux vecteurs.

Proposition 8.5. *Si* $W^1, \ldots, W^q$ *sont des matrices fractales de même dimensions alors* $\left[W^1/\ldots/W^q\right]$ *et* $\left[W^1|\ldots|W^q\right]$ *sont aussi des matrices fractales.*

En utilisant ces opérateurs, on peut alors réécrire le système précédent.

Proposition 8.6. *En utilisant les notations précédentes, on a* $W_{n1,\ldots,n_k}.\overrightarrow{F}_{c_{inv}} = 0$ *ssi :*

$$\left[\ [0|0|B]/\quad [0|A|0]/\quad [D|0|0]/\quad [A|D|0]/\quad [B|B|A+D]\ \right].\left[\ F_X/\quad F_{!X}/\quad F_{All}\ \right] = 0$$

Démonstration. Conséquence directe de la définition des opérateurs. □

Nous proposons maintenant un algorithme pour le calcul d'un famille génératrice de flots positifs simples.
Cet algorithme prend comme entrée soit une matrice entière et un ensemble de paramètres réduit à l'ensemble vide (système non paramétré), soit une matrice fractale en bloc avec un ensemble de paramètres compatibles avec W (la taille de W est $n_1 \times n_1$ et chaque élément est soit une matrice entière soit une fractale de taille $n_2 \times n_2$ et ainsi de suite). La sortie de cet algorithme est soit un ensemble de vecteurs entiers (lorsque les paramètres sont réduits à l'ensemble vide) soit un ensemble de sommes formelles $F_{<>} = \langle X_1, X_2, \ldots, X_k\rangle f_1 + \langle X_1, X_2, \ldots, !X_k\rangle f_2 + \langle X_1, X_2, \ldots, All_k\rangle f_3 + \ldots + \langle All_1, All_2, \ldots, All_k\rangle f_{3^k}$ avec f_i des vecteurs entiers sur P qui génèrent des flots positifs simples.
Algorithme 1 : Simple_Positive_Solutions(W, Parameters = $\{n_1, \ldots, n_k\}$)

If (Parameters = $\emptyset$) Then

+ return $\{X|W.X = 0\}$ – *integer vectors computed with the Farkas algorithm*

Else

+ Construct the **fractal** matrix W' defined by [1]

$$W' = \left[\ [0|0|B]/\quad [0|A|0]/\quad [D|0|0]/\quad [A|D|0]/\quad [B|B|A+D]\ \right]$$

+ Compute the set SF of solutions of the system $W'.\left[\ F_X/\quad F_{!X}/\quad F_{All}\ \right] = 0$. with this algorithm : SF := Simple_Positive_Flows(W',$\{n_2, \ldots, n_k\}$) [2]
+ Return the set of formal sums

$$\left\{F = X_{C_1}.F_X + All_{C_1}.F_{All} + !X_{C_1}.F_{!X}, \left[\ F_X/\quad F_{!X}/\quad F_{All}\ \right] \in SF\right\}$$

End if ;

Proposition 8.7. *L'ensemble calculé par le précédent algorithme définit une famille génératrice de flots positifs simples d'un réseau SWF.*

Démonstration. Par récurrence sur l'ensemble des paramètres :

1. si Parameters = $\emptyset$ alors l'ensemble calculé est une famille génératrice puisque nous utilisons l'algorithme de Farkas ;
2. Supposons qu'étant donné une matrice fractale W' et un ensemble compatible $\{n_2, \ldots, n_k\}$ l'algorithme précédent calcule un famille génératrice de flots positifs simples.

[1] comme précédemment, nous notons A, B et D les blocs de la matrice fractale W
[2] si $k < 2$, alors $\{n_2, \ldots, n_k\} = \emptyset$

(a) les sommes formelles calculées par l'algorithme définissent des flots positifs simples du réseau défini par W. De fait, l'hypothèse de récurrence combinée avec la proposition 8.6 assure que nous calculons effectivement des flots positifs simples.

(b) l'ensemble est générateur. De fait, soit $\mathcal{F}0$ un flot positif simple, en utilisant la proposition 8.3, toute interprétation $\overrightarrow{F0}_{c_{inv}}$ de $\mathcal{F}0$ peut être écrite avec $F0_X$, $F0_{All}$ et $F0_{!X}$ comme défini dans cette proposition. En utilisant la proposition 8.6, on obtient que $W'.\left[\begin{array}{ccc} F0_X/ & F0_{!X}/ & F0_{All} \end{array}\right] = 0$. Par hypothèse de récurrence, comme nous calculons un famille génératrice de solutions de $W'.\left[\begin{array}{ccc} F_X/ & F_{!X}/ & F_{All} \end{array}\right] = 0$ alors $\left[\begin{array}{ccc} F0_X/ & F0_{!X}/ & F0_{All} \end{array}\right]$ est généré par cet ensemble et tout interprétation de $\mathcal{F}0$ est générateur de l'ensemble calculé. Ainsi, les sommes formelles calculées par le précédent algorithme génèrent tous les flots positifs simples du réseau SWF défini par la matrice fractale W.

□

Si l'on note $K_{P\times T}$ la complexité de l'algorithme de Farkas pour un réseau de P places et T transitions, alors la complexité de l'algorithme précédent est $K_{3^k.P\times 5^k.T}$ avec k le nombre de classes du réseau. Cependant, comme les matrices créées par l'algorithme sont très creuses, les premiers résultats obtenus tendent à prouver que l'algorithme se comporte plutôt comme si la complexité était $2^k.K_{P\times T}$ ce qui reste un assez bonne complexité puisque même pour certains modèles complexes, k peut rester assez faible.

d. Traiter les réseaux SWF non homogènes

Supposons maintenant que le réseau ne soit pas homogène. Afin de pouvoir utiliser l'algorithme précédent, il est nécessaire d'homogénéiser le réseau. Deux cas doivent être considérés : 1) une transition possède un domaine de couleur plus grand que ses places adjacentes (le contraire n'est pas possible étant donné la définition des réseaux SWF) ; 2) un *mapping* coloré utilise un valeur constante ou un *mapping* arbitraire.. Afin de rendre le réseau homogène, nous procédons en deux étapes :

1. dès qu'un *mapping* constant (d'une classe C) ou un *mapping* arbitraire apparaît sur un arc, nous replaçons tous les *mappings* de cette classe par All_C ; ce remplacement aboutit à la perte de la synchronisation et le réseau obtenu effectue une simulation faible de l'original. Ainsi, tous les flots calculés dans ce réseau sont également des flots du réseau original [1]. De plus, comme nous interdisons le mixage d'un *mapping* constant et de All_C dans la définition des réseaux SWF, il nous faut fixer la taille du paramètre de C (ce qui sera nécessaire si nous avons besoin d'homogénéiser un arc avec le *mapping* All_C puisqu'il sera remplacé par $|C|.All_C$).
2. calculer le plus petit commun multiple (C_{lcm}) des domaines de couleurs (en étendant la multiplication et la division au produit des classes) et étendre le domaine de couleur de chaque place et transition de manière à ce que leur domaine de couleur soit C_{lcm}. Modifier ensuite le marquage initial en fonction de C_{lcm}. Par exemple, le *lcm* de $C_1 \times C_1 \times C_2$, $C_1 \times C_2 \times C_3$ et $C3 \times C_3$ est $C_1 \times C_1 \times C2 \times C_3 \times C_3$. Supprimer dans chaque flot obtenu les couleurs ajoutées lors de l'homogénéisation. Comme ces transformations ne modifient pas le comportement du modèle, il est clair que les flots positifs calculés de cette manière sont ceux du modèle original.

Par exemple, considérons le simple réseau bien formé de la figure 8.4 qui modélise une affectation atomique `Free := f(Id, X)` où f est une fonction booléenne arbitraire et où la place `Write` modélise un verrou de type lecteur/rédacteur. La première étape (homogénéisation de *mappings* constants et arbitraires) produit le modèle de la figure 8.5.
Le plus petit commun domaine de couleur est $C\_Id \times C\_Int \times C\_Bool$. Après homogénéisation du réseau, on obtient le modèle de la figure 8.6. ON peut alors remarquer que l'information concernant

[1] Il est possible de traitement plus finement les *mappings* constants.

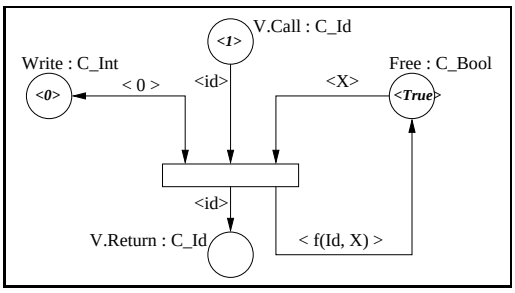


FIG. 8.4 – Un réseau simple

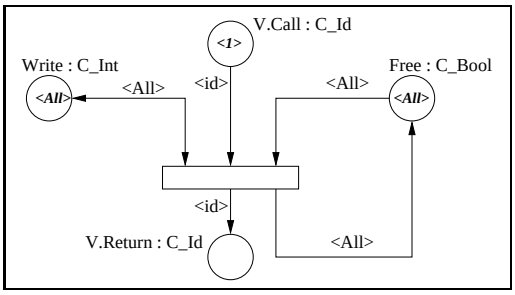


FIG. 8.5 – Le réseau précédent suite à une première homogénéisation

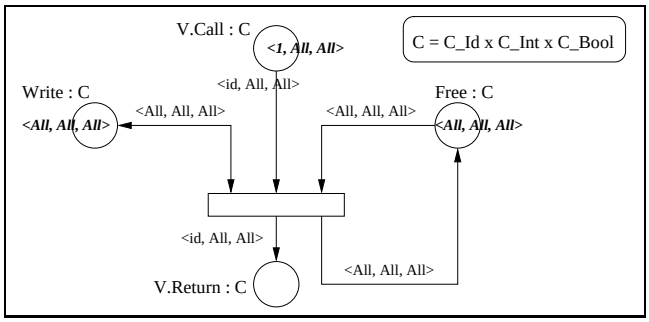


FIG. 8.6 – Le réseau précédent homogénéisé

le domaine de couleur *C_Bool* a été oublié par le processus d'homogénéisation (et que le modèle est moins lisible).

Une fois l'homogénéisation effectuée, on peut alors calculer les flots positifs : par exemple, la somme $\langle X, All, All\rangle.(V.Call + V.Return)$ défini un flot positif simple du dernier modèle. Pour obtenir le flot positif correspondant sur le modèle original, il suffit de supprimer la partie colorée ajoutée pour l'homogénéisation : pour le précédent invariant, on obtient le flot positif correct : $\langle X\rangle.(V.Call + V.Return)$.

e. Exemple

Considérons de nouveau le réseau de la figure 8.3. Sa matrice d'incidence après homogénéisation est :

$$W = \begin{array}{c} \\ TakeChairs \\ takeL \\ takeR \\ giveL \\ giveR \\ end \end{array} \begin{array}{c} \begin{array}{cccccccc} Thinking & Att1 & Att2 & Eating & Ending1 & Ending2 & Forks & Chairs \end{array} \\ \begin{pmatrix} -\langle X\rangle & \langle X\rangle & 0 & 0 & 0 & 0 & 0 & -\langle All\rangle \\ 0 & -\langle X\rangle & \langle X\rangle & 0 & 0 & 0 & -\langle X\rangle & 0 \\ 0 & 0 & -\langle X\rangle & \langle X\rangle & 0 & 0 & -\langle !X\rangle & \langle All\rangle \\ 0 & 0 & 0 & -\langle X\rangle & \langle X\rangle & 0 & \langle X\rangle & 0 \\ 0 & 0 & 0 & 0 & -\langle X\rangle & \langle X\rangle & \langle !X\rangle & 0 \\ \langle X\rangle & 0 & 0 & 0 & 0 & -\langle X\rangle & 0 & 0 \end{pmatrix} \end{array}$$

Les matrices A, B et D correspondantes sont :

$$A = \begin{array}{c} \\ TakeChairs \\ takeL \\ takeR \\ giveL \\ giveR \\ end \end{array} \begin{array}{c} \begin{array}{cccccccc} Thinking & Att1 & Att2 & Eating & Ending1 & Ending2 & Forks & Chairs \end{array} \\ \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{pmatrix} \end{array}$$

$$B = \begin{array}{c} \\ TakeChairs \\ takeL \\ takeR \\ giveL \\ giveR \\ end \end{array} \begin{array}{c} \begin{array}{cccccccc} Thinking & Att1 & Att2 & Eating & Ending1 & Ending2 & Forks & Chairs \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$D = \begin{array}{c} \\ TakeChairs \\ takeL \\ takeR \\ giveL \\ giveR \\ end \end{array} \begin{array}{c} \begin{array}{cccccccc} Thinking & Att1 & Att2 & Eating & Ending1 & Ending2 & Forks & Chairs \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

En appliquant l'algorithme présenté à la page 202 on aboutit à 6 flots :
- $\mathcal{F}_1 = \langle X\rangle.Thinking + \langle X\rangle.Att1 + \langle X\rangle.Att2 + \langle X\rangle.Eating + \langle X\rangle.Ending1 + \langle X\rangle.Ending2$
- $\mathcal{F}'_1 = \langle !X\rangle.Thinking + \langle !X\rangle.Att1 + \langle !X\rangle.Att2 + \langle !X\rangle.Eating + \langle !X\rangle.Ending1 + \langle !X\rangle.Ending2$
- $\mathcal{F}''_1 = \langle All\rangle.Thinking + \langle All\rangle.Att1 + \langle All\rangle.Att2 + \langle All\rangle.Eating + \langle All\rangle.Ending1 + \langle All\rangle.Ending2$
- $\mathcal{F}_2 = \langle !X\rangle.Forks + \langle !X\rangle.Att2 + \langle !X\rangle.Eating + \langle X\rangle.Eating + \langle X\rangle.Ending1$
- $\mathcal{F}_3 = \langle All\rangle.Att1 + \langle All\rangle.Att2 + \langle X\rangle Chairs$ $\mathcal{F}'_3 = \langle All\rangle.Att1 + \langle All\rangle.Att2 + \langle X!\rangle Chairs$

Cet exemple met en évident le fait que, dans un certain nombre de cas, notre algorithme calcule un famille génératrice de tous les flots positifs. Cependant, il souligne également deux difficultés associées à notre méthode. Premièrement, nous calculons un certain nombre de flots inutiles tels que $\mathcal{F}'_1$, $\mathcal{F}''_1$ ou $\mathcal{F}'_3$; en effet, dès que $\langle X_C\rangle.F$ est un flot $\langle !X_C\rangle.F$ et $\langle All\rangle.F$ sont également deux flots et si $\langle X_C\rangle.F + \langle !X_C\rangle.F'$ est un flot alors $\langle All\rangle.(F + F')$ est également un flot. Nos expérimentations montrent que nous calculons en général un flot inutile par flot et par domaine de couleur.
Comme la complexité de l'algorithme de Farkas dépend principalement du nombre de solutions, notre méthode se comporte comme si nous calculions des flots positifs dans un réseau deux fois

plus grand que l'original. Nous souhaitons alors étudier la possibilité d'utiliser des heuristiques pour résoudre ce léger problème.
Le second problème est que, par définition, certains flots positifs ne peuvent être calculés. Par exemple, un flot faisant intervenir trois couleurs différentes (X, $X!$ and $X!!$) ou un flot utilisant le cardinal d'une classe comme poid ($n_1.\langle X\rangle.F$) ne sont pas des flots simples et ainsi, ne sont pas calculés.

8.3 Conclusion

La validation et la spécification de programmes concurrents est un problème complexe. Peu d'outils basés sur les réseaux colorés ont été conçus pour répondre à ce besoin dans la mesure où la majorité de ces outils se concentrent sur la représentation du contrôle plutôt que des données, p.ex., les outils basés sur les réseaux bien formés tels que GreatSPN [107]. De par ses fonctionnalités, Helena peut répondre à ce besoin.

La possibilité de définir des types de données de haut niveau ainsi que des fonctions complexes écrites dans un langage de programmation inspiré d'Ada permet dans une large mesure de modéliser des programmes concurrents ou d'extraire aisément des réseaux colorés de programmes concurrents.

La possibilité de répartir l'exploration et le stockage de l'espace d'état nous permet alors de franchir un cap dans la taille des modèle que l'on peut analyser.

Les travaux futurs ont, pour la plupart, comme objectif d'essayer de franchir de nouveaux paliers dans les possibilités d'analyse. Ainsi, il nous paraît important d'essayer d'enrichir notre outil en offrant la possibilité de stocker l'espace d'état sur le disque dur ce qui nous permettrait de bénéficier de ressources mémoires bien plus importantes.

Une seconde perspective concerne le passage à l'échelle. Cet aspect n'a pas été considéré lors de cette thèse. Le passage à l'échelle pose de nouvelles problématiques que nombre de nos solutions ne sont à priori pas capable de gérer efficacement. Obtenir un degré de localité intéressant à large échelle semble difficile avec les méthodes que nous utilisons. Seule la redondance et la duplication d'état semblent en mesure de garantir un certain degré de localité.
De même, les méthodes présentées pour l'adaptation des méthodes d'ordre partiel semblent inadéquates. Que ce soit notre *proviso* ou celui de Spin, leur utilisation à large échelle sera vraisemblablement inefficace car générera énormément de communications. Là encore, seule la redondance et la duplication d'état pourraient permettre d'utiliser les méthodes d'ordre partiel de façon très locale.

Cyclades – tout comme Helena – étant pour l'instant limité à la vérification de propriété d'accessibilité, il nous paraît important d'intégrer la possibilité de valider des propriétés plus complexes (par exemple des propriétés LTL).

Conclusion et perspectives

Dans cette thèse, nous avons proposé des techniques de vérification automatiques pour les réseaux de Petri de haut-niveau. Ces techniques s'attaquent essentiellement au problème de l'explosion combinatoire en essayant de tirer parti d'un réseau de machines. Nous avons alors essayé d'adapter les techniques visant à contenir l'explosion de l'espace d'état que ce soit *via* une représentation efficace des états, ou au moyen de méthodes de réduction de l'espace d'état.
D'un point de vue théorique, nos contributions sont les suivantes :

- nous avons proposé des méthodes de partitionnement de l'espace d'état pour les réseaux de Petri colorés. Ces méthodes reposent sur une analyse structurelle pour essayer d'obtenir une répartition offrant un degré de localité suffisant pour permettre une adaptation efficace des méthodes de représentation d'état ou de réduction de l'espace d'état.
- les algorithmes de réduction de l'espace d'état sont tous totalement liés au parcours de l'espace d'état et plus particulièrement à sa cohérence globale. Cette cohérence ne peut plus être garantie dès lors que le parcours de l'espace d'état est parallélisé. Nous avons alors proposé un nouvel algorithme de réduction de l'espace d'état visant à relâcher les contraintes de cohérences sur le parcours. Cette méthode fonctionne très efficacement en séquentiel, et dans de nombreux cas mieux que les principales méthodes existantes. Cette méthode fonctionne également particulièrement bien lors d'une exploration parallélisée. Elle permet notamment d'effectuer de très bonnes réductions sans avoir à effectuer une grand quantité d'explorations redondantes.
- l'analyse structurelle de réseaux de Petri colorés peut permettre l'amélioration de nombreuses techniques visant à combattre le problème de l'explosion combinatoire. L'algorithme de calcul semi-flots colorés pourrait alors permettre d'améliorer les méthodes de réductions structurelles et les méthodes d'ordre partiel. En nous donnant des informations comportementales, nous espérons qu'il sera possible de partitionner l'espace d'état de façon plus efficace.

D'un point de vue pratique, ces travaux ont aboutit à l'implémentation du model checker réparti et parallèle Cyclades. Il nous a notamment permis de valider nos approches théoriques. Il peut également s'intégrer parfaitement dans la plate-forme Quasar puisqu'il utilise le même langage de spécification que l'outil Helena.

Perspectives

Les travaux de recherches dans le cadre du model checking réparti et parallèle sont nombreuses et assez diversifiées.

Un premier objectif concerne bien évidemment la vérification de propriété de logique temporelles. Plusieurs travaux ont déjà été menés dans le cadre de vérification de propriétés LTL sur des architectures à mémoire partagée et à mémoire répartie.

Une deuxième perspective – dont nous avons déjà parlé (5.7) – concerne la version parallèle et répartie de Cyclades. Il serait intéressant d'ajouter la possibilité de lancer plusieurs stratégies de partitionnement en parallèle. Chaque *thread* d'un même nœud utilisant une fonction de partition spécifique. En analysant, à l'éxecution, les résultats obtenus pour les divers partitionnements, on pourrait alors abandonner certaines approches pour se concentrer celles qui présentent de meilleurs

résultats en leur allouant plus de *threads*. Cette approche plus fine pourrait alors offrir de meilleurs résultats.

Enfin, comme nous l'avons fait pour le portage d'Helena ou de Cyclades sur des architectures parallèles, il serait intéressant de se pencher sur le stockage de l'espace sur disque [71]. L'arrivée massive des nouvelles architectures parallèles à changé notre manière de considérer les systèmes multithreadés. Avant cette démocratisation, la parallélisation d'une application résultait d'un véritable besoin. Aujourd'hui la démarche n'est plus la même. Le raisonnement "*Mon application nécessite d'effectuer un très grand nombre de calculs et peine à s'éxecuter dans des temps raisonnables sur une machine mono-processeur classique, il faudrait peut-être que je cherche à acquérir une machine multi-processeurs pour y paralléliser mon application.*" tend à être remplacé par le raisonnement suivant : "*J'ai plusieurs processeurs disponibles mais un seul travaille. Plutôt que de gâcher des ressources, est-ce que je ne pourrais pas paralléliser mon application et en retirer un gain de temps intéressant ?*".

Aujourd'hui encore, le disque dur est considéré – à raison – comme le point noir des machines actuelles. Son très faible débit n'est pas compensé par les énormes capacités de stockage. Le stockage de données sur disque n'est donc utilisé qu'en dernier recours avec des dégradations en temps potentiellement importantes. L'apparition de disques durs basés sur la technologie *flash* (disques SDD – *Solid State Drive*) pourrait bien changer la donne et relancer l'utilisation du disque dur dans le model checking comme dans d'autres domaines. L'accroissement des vitesses des transferts de données sur ces nouveaux disques est particulièrement intéressant puisqu'il pourra alors nous offrir de formidables espaces de stockages à des coûts fortement réduits. Si l'on envisage alors la possibilité d'utiliser plusieurs disques durs (locaux à chacun des nœuds) en parallèle, les possibilités sont gigantesques. Aujourd'hui, l'espace disponible sur un disque dur est généralement plus de 100 fois supérieur à celui proposé par la mémoire vive. On pourrait alors envisager la vérification de systèmes beaucoup plus conséquents.
Nous avons vu dans cette thèse qu'il était possible de conserver une certaine efficacité des méthodes séquentielles dans un contexte réparti où quelques dizaines de machines sont utilisées. En essayant d'obtenir un bon degré de localité, nous avons pu approcher l'efficacité obtenue en séquentiel de méthodes telles que les Δ-*markings* ou les réductions d'ordre partiel. Cependant, en utilisant quelques dizaines de machines, le gain en ressource mémoire que l'on peut espérer n'est que d'une puissance de dix. Pour les modèles paramétrés, une exploration effectuée en séquentiel nous permet d'atteindre une valeur n maximale pour le paramètre. Dans certains cas, l'explosion combinatoire est telle que la puissance de dix obtenue suite à la répartition ne nous permet pas de valider le modèle avec la valeur $n+1$ du paramètre. La seule solution nous permettant de gagner une nouvelle puissance de dix est donc de passer à une ou plusieurs centaines de machines. Dans ce cas là, le degré de localité sera fortement réduit et l'efficacité de certaines méthodes pourrait alors en être fortement réduite.
Les capacités de stockage des disques durs, généralement plus de 100 fois supérieures à la capacité de la mémoire nous permet alors de gagner deux puissances de dix. Couplé à une exploration répartie, le gain est alors de trois puissances de dix. Les temps d'accès des disques durs SDD étant alors considérablement réduite et les capacités augmentant alors fortement, ce couplage de stratégies pourraient alors être véritablement intéressant.

Une dernière perspective évidente concerne le passage à l'échelle. La question du passage à l'échelle concerne cependant l'aspect réparti beaucoup plus que l'aspect parallèle. Dans ce premier cas, les problématiques sont à priori véritablement différentes de celles abordées dans cette thèse. Plusieurs approches semblent totalement inadaptée au passage à l'échelle et notamment les réductions d'ordre partiel. D'une manière générale, le passage à l'échelle ne peut qu'être difficillement envisageable sans un degré de localité raisonnable. La duplication et l'exploration redondante semblent donc plus ou moins inévitables. Cette duplication pose alors d'autres problèmes : doit-on supprimer des états régulièrement (en utilisant un mécanisme de *Garbage Collector* par exemple) ? Si oui, ces suppressions ne poseront-elles pas des problèmes de terminaison ?

Concernant l'outil Quasar, la perspective à moyen et long terme consiste à essayer d'étendre les possibilités de modélisation en certains éléments des langages de programmation de haut niveau. A plus court terme, il nous paraît indispensable de tester Quasar sur un grand nombre de programmes de taille conséquentes.

Index

Bibliographie

Réseaux de Petri

[1] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. On well-formed coloured nets and their symbolic reachability graph. In *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, pages 373–396. Springer-Verlag, 1991.

[2] Michel Diaz. *Les Réseaux de Petri - Modèles Fondamentaux*, volume 1. HERMES Sciences Publications, 2001.

[3] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets. *Bulletin of the EATCS*, 52 :244–262, 1994.

[4] J. Ezpeleta, F. García-Vallés, and José Manuel Colom. A class of well structured petri nets for flexible manufacturing systems. In *ICATPN '98 : Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, pages 64–83, London, UK, 1998. Springer-Verlag.

[5] Serge Haddad. *Une catégorie régulière de réseaux de Petri de haut niveau : définition, propriétés et réductions. Application à la validation de systèmes distribués.* PhD thesis, Université Paris 6, 1987.

[6] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in coloured Petri nets. In Grzegorz Rozenberg, editor, *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 313–341. Springer-Verlag, 1991.

[7] Kurt Jensen. Coloured Petri nets and the invariant method. *Theor. Comp. Science 14*, pages 317–336, 1981.

[8] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1, Basic Concepts. Springer-Verlag, 1997.

[9] Tadao Murata. Petri nets : Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, 1989.

Analyse structurelle

[10] Gérard Berthelot. Checking properties of nets using transformation. In *Proceedings of the 6th European Workshop Applications and Theory in Petri Nets.*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1985.

[11] J. M. Colom and M. Silva. Convex geometry and semiflows in P/T nets : a comparative study of algorithms for computation of minimal p-semiflows. In *APN 90 : Proceedings on Advances in Petri nets 1990*, pages 79–112, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[12] José-Manuel Colom and Fernando Garcia-Valles. Implicit places in net systems. In *Proceedings of the 8th International Workshop on Petri Net and Performance Models*, pages 104–113. IEEE Press, 1999.

[13] J. M. Couvreur, S. Haddad, and J. F. Peyre. Computation of generative families of positive semi-flows in two types of coloured nets. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, 1991, Gjern, Denmark*, pages 122–144, June 1991. NewsletterInfo : 39.

[14] J. M. Couvreur, S. Haddad, and J. F. Peyre. Generative families of positive invariants in coloured nets sub-classes. *Lecture Notes in Computer Science ; Advances in Petri Nets 1993*, 674 :51–70, 1993.

[15] J.M. Couvreur. The general computation of flows for coloured nets. In *proc of the 11th International Conference on Application and Theory of Petri-Nets*, Paris, France, June 1990.

[16] J.M. Couvreur and S. Haddad. Towards a general and powerful computation of flows for parameterized coloured nets. In *9th European Workshop on Application and Theory of Petri Nets*, volume II, Venice (Italy), June 1988.

[17] Sami Evangelista, Serge Haddad, and Jean-François Pradat-Peyre. New coloured reductions for software validation. In *Proccedings of the 7th International workshop on discrete event systems*, pages 355–360, 2004.

[18] Sami Evangelista, Serge Haddad, and Jean-François Pradat-Peyre. Syntactical colored Petri nets reductions. In *Proccedings of the Third International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 202–216. Springer-Verlag, 2005.

[19] Sami Evangelista, Christophe Pajault, and Jean-François Pradat-Peyre. A simple positive flows computation algorithm for a large subclass of colored nets. In *FORTE*, pages 177–195, 2007.

[20] H. J. Genrich and K. Lautenbach. S-invariance in predicate/transition nets. In Pagnoni, A. and Rozenberg, G., editors, *Informatik-Fachberichte 66 : Application and Theory of Petri Nets — Selected Papers from the Third European Workshop on Application and Theory of Petri Nets, Varenna, Italy, September 27–30, 1982*, pages 98–111. Springer-Verlag, 1983.

[21] S. Haddad and C. Girault. Algebraic structure of flows of a regular coloured net. *Lecture Notes in Computer Science : Advances in Petri Nets 1987*, 266 :73–88, 1987. NewsletterInfo : 27.

[22] Serge Haddad and Jean-François Pradat-Peyre. New efficient Petri nets reductions for parallel programs verification. *Parallel Processing Letters*, 1 :16, 2006.

[23] G. Memmi and J. Vautherin. Computation of flows for unary-predicates/transition nets. *Lecture Notes in Computer Science : Advances in Petri Nets 1984*, 188 :455–467, 1985.

[24] Denis Poitrenaud and Jean-François Pradat-Peyre. Pre- and post-agglomerations for LTL model checking. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, volume 1825 of *Lectures Notes in Computer Science*, pages 387–408. Springer-Verlag, 2000.

[25] W. Reisig. *EATCS-An Introduction to Petri Nets*. Springer-Verlag, 1983.

[26] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80 :1–34, 1991. NewsletterInfo : 38,39.

[27] M. Silva Suarez, J. Martinez, P. Ladet, and H. Alla. Generalized inverses and the calculation of symbolic invariants for colored petri nets. *Technique et Science Informatiques*, 4(1) :113–126, 1985. NewsletterInfo : 16,21,22.

[28] J. Vautherin. Calculation of semi-flows for pr/T-systems. In *Int. Workshop on Petri Nets and Performance Models, Madison, Wisconsin*, pages 174–183, Washington, 1987. IEEE Computer Society Press. NewsletterInfo : 29.

Model checking séquentiel

[29] G. Chiola, G. Franceschinis, and R. Gaeta. A symbolic simulation mechanism for well-formed coloured petri nets. In *ANSS '92 : Proceedings of the 25th annual symposium on Simulation*, pages 192–201, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[30] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[31] Sami Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut niveau. Application à la vérification de programmes Ada concurrents.* PhD thesis, Conservatoire National des Arts et Métiers, Paris, 2006.

Représentation de l'espace d'état

[32] Sami Evangelista and Jean-François Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2005.

[33] Jaco Geldenhuys. State caching reconsidered. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 23–38. Springer-Verlag, 2004.

[34] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191. Springer-Verlag, 1992.

[35] Gerard J. Holzmann. State compression in spin : Recursive indexing and compression training runs. In *Proceedings of the Third Spin Workshop*, 1997.

Réductions d'ordre partiel

[36] T. Basten and D. Bosnacki. Enhancing partial-order reduction via process clustering. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 245–253. IEEE Computer Society, 2001.

[37] D. Bosnacki, S. Leue, and A. Lluch-Lafuente. Partial-order reduction for general state exploring algorithms. In *Proceedings of the 13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 271–287. Springer-Verlag, 2006.

[38] Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 91–105. Springer-Verlag, 2005.

[39] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, pages 279–287, 1999.

[40] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2-3) :247–267, 2004.

[41] Sami Evangelista and Christophe Pajault. Some solutions to the ignoring problem. *Lecture Notes in Computer Science*, pages 76–94, 2007.

[42] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121. ACM, 2005.

[43] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[44] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems.* PhD thesis, University of Liege, Computer Science Department, 1994.

[45] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third International Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342. Springer-Verlag, 1991.

[46] Patrice Godefroid and Pierre Wolper. Partial-order methods for temporal verification. In *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246. Springer-Verlag, 1993.

[47] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 197–211, 1994.

[48] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *TACAS '98 : Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

[49] Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Form. Methods Syst. Des.*, 20(3) :231–247, 2002.

[50] Doron Peled. All from one, one for all : on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.

[51] Pierre-Olivier Ribet, François Vernadat, and Bernard Berthomieu. Graphe de pas persistant. In *Journées FAC'2002. Formalisation des Activités Concurrentes*, page 15p., 2002.

[52] Pierre-Olivier Ribet, François Vernadat, and Bernard Berthomieu. On combinining the persistent sets method with the covering steps graph method. In *Proceedings of the 22th International Conference on Formal Methods for Networked and Distributed Systems*, volume 2529 of *Lecture Notes in Computer Science*, pages 344–359. Springer-Verlag, 2002.

[53] Antti Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[54] Antti Valmari. *State Space Generation : Efficiency and Practicality.* PhD thesis, Tampere University of Technology, 1988.

[55] Antti Valmari. Eliminating redundant interleavings during concurrent program verification. In *Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1989.

[56] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the second International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, 1990.

[57] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515, Berlin, Germany, 1991. Springer-Verlag.

[58] Antti Valmari. On-the-fly verification with stubborn sets. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[59] Antti Valmari. State of the art report : Stubborn sets. *Petri Net Newsletter*, pages 6–14, 1994.

[60] Kimmo Varpaaniemi. Efficient detection of deadlock in Petri nets. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1993.

[61] Kimmo Varpaaniemi. On combining the stubborn set method with the sleep set method. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 548–567. Springer-Verlag, 1994.

[62] Kimmo Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation.* A51, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1998.

[63] François Vernadat, Pierre Azéma, and François Michel. Covering step graph. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, pages 516–535. Springer-Verlag, 1996.

[64] François Vernadat and François Michel. Covering step graph preserving failure semantics. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 253–270. Springer-Verlag, 1997.

Model-Checking symbolique

[65] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 340–351. Springer-Verlag, 1997.

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

[67] Jerry R. Burch, Edmund M. Clarke, David L. Dill, Hwang L.J., and Ken McMillan. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[68] Ken McMillan. *Symbolic Model Checking.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

Model-Checking parallèle et réparti

[69] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *CAV '00 : Proceedings of the 12th International Conference on Computer Aided Verification*, pages 216–231, London, UK, 2000. Springer-Verlag.

[70] Alexander Bell and Boudewijn R. Haverkort. Sequential and distributed model checking of petri net specifications. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[71] Alexander Bell and Boudewijn R. Haverkort. Distributed disk-based algorithms for model checking very large markov chains. *Form. Methods Syst. Des.*, 29(2) :177–196, 2006.

[72] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free μ-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*. Springer-Verlag Inc., 2002.

[73] Victor Braberman, Alfredo Olivero, and Fernando Schapachnik. Issues in distributed timed model checking : Building zeus. *Int. J. Softw. Tools Technol. Transf.*, 7(1) :4–18, 2005.

[74] William J. Knottenbelt, Mark Mestern, Peter G. Harrison, and Pieter S. Kritzinger. Probability, parallelism and the state space exploration problem. In *Computer Performance Evaluation (Tools)*, pages 165–179, 1998.

[75] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in Eddy. In *SPIN*, pages 108–125, 2006.

Partitionnement de l'espace d'états

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

[77] Gianfranco Ciardo and David M. Nicol. Automated parallelization of discrete state-space generation. Technical Report Technical Report NASA/CR-2000-210082, NASA Langely Research Center, Hampton, USA, 2000.

[78] R. Kumar, M. D. Jones, and E. G. Mercer. Dynamic partition algorithm for distributed murphi. Technical Report VV-0402 (also VV-03), Dept. of Computer Science, Brigham Young U., 2004.

[79] R. Kumar and E.G. Mercer. Load balancing parallel explicit state model checking. In *Parallel and Distributed Model Checking*, London, UK, August 2004.

[80] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, London, UK, 1999. Springer-Verlag.

[81] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01 : Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[82] Laure Petrucci and Lars Michael Kristensen. An approach to distributed state space exploration for coloured petri nets. In *25th International Conference on Application and Theory of Petri Nets*, pages 474–483. Springer-Verlag, 2004.

[83] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In *CAV '97 : Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278, London, UK, 1997. Springer-Verlag.

Réductions d'ordre partiel

[84] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed Partial Order Reduction of State Spaces. In *3rd International Workshop on Parallel and Distributed Methods in verifiCation*, Sep. 2004.

[85] R. Palmer and G. Gopalakrishnan. Partial order reduction assisted parallel model checking. In *Proc. Parallel and Distributed Model Checking (PDMC) Workshop*, 2002.

Model Checking parallèle

[86] S. C. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady–state solution of gspns on a shared–memory multiprocessor. In *PNPM '97 : Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 112, Washington, DC, USA, 1997. IEEE Computer Society.

[87] J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core LTL model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[88] Stefano Caselli, Gianni Conte, and P. Marenzoni. Parallel state space exploration for gspn models. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 181–200, London, UK, 1995. Springer-Verlag.

[89] Gerard J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electron. Notes Theor. Comput. Sci.*, 198(1) :3–16, 2008.

[90] G.J. Holzmann and Dragan Bosnacki. Multi-core model checking with Spin. In *HIPS-TopModels 2007, Long Beach*, March 26-30 2007.

Vérification de propriétés LTL

[91] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Brno : Faculty of Informatics, Masaryk University Brno, 2004.

[92] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. Parallel breadth-first search ltl model-checking. In *ASE*, pages 106–115, 2003.

[93] L. Brim and J. Barnat. Distribution of explicit-state LTL model-checking. In Thomas Arts and Wan Fokkink, editors, *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[94] Luboš Brim, Ivana Černá, Pavel Krčál, and Radek Pelánek. Distributed LTL model checking based on negative cycle detection. *Lecture Notes in Computer Science*, 2245 :96–107, 2001.

[95] L. Brim J. Barnat and J. Stríbrná. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 200–216. Springer-Verlag New York, Inc., 2001.

Model checking symbolique

[96] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. In *FMCAD '00 : Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 390–404, London, UK, 2000. Springer-Verlag.

[97] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Form. Methods Syst. Des.*, 29(2) :157–175, 2006.

[98] Tamir Heyman, Daniel Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV*, pages 20–35, 2000.

Analyse de programmes concurrents

[99] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[100] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[101] C. Kaiser, C. Pajault, and J.-F. Pradat-Peyre. Modelling remote concurrency with ada. case study of symmetric non-deterministic rendez-vous. In *Proceedings 12th International Conference on Reliable Software Technologies*, volume 4498 of *Lecture Notes in Computer Science*, pages 192–207. Springer-Verlag, 2007.

[102] C. Kaiser, C. Pajault, and J.-F. Pradat-Peyre. Concurrent program metrics drawn by Quasar. In *13th Int. Conf on Reliable Software Technologie*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[103] Pierre Rousseau. *Découpe de programmes concurrents en vue de leur vérification*. PhD thesis, Conservatoire National des Arts et Métiers, Paris, 2006.

[104] Pierre Rousseau. A new approach for concurrent program slicing. In *Proceedings of the 26th International Conference on Formal Methods for Networked and Distributed Systems*, volume 4229 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, 2006.

Outils

[105] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing : A model checker for concurrent software. Technical report, Microsoft Research, Microsoft Corporation, http ://www.research.microsoft.com, 2004.

[106] Thomas Ball and Sriram K. Rajamani. The slam project : debugging system software via static analysis. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[107] Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, and Marina Ribaudo. Greatspn 1.7 : Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1-2) :47–68, 1995.

[108] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, volume 2404 of *Lectures Notes in Computer Science*, pages 359–364. Springer-Verlag, 2002.

[109] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin : A dynamic extension of spin. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software*, volume 1680 of *Lectures Notes in Computer Science*, pages 261–276. Springer-Verlag, 1999.

[110] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with hsf-spin. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *Lectures Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.

[111] Sami Evangelista. High level Petri nets analysis with Helena. In *Proceedings of the 25th International Conference on Application and Theory of Petri Nets*, volume 3536 of *Lecture Notes in Computer Science*, pages 455–464. Springer-Verlag, 2005.

[112] Sami Evangelista, Claude Kaiser, Christophe Pajault, Jean-François Pradat-Peyre, and Pierre Rousseau. Dynamic tasks verification with quasar. In *Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies*, volume 3555 of *Lecture Notes in Computer Science*, pages 91–104. Springer-Verlag, 2005.

[113] Sami Evangelista, Claude Kaiser, Jean-François Pradat-Peyre, and Pierre Rousseau. Quasar : a new tool for analysing concurrent programs. In *Proceedings of the 8th Ada-Europe International Conference on Reliable Software Technologies*, volume 2655 of *Lecture Notes in Computer Science*, pages 168–181. Springer-Verlag, 2003.

[114] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[115] Patrice Godefroid. Software model checking : The verisoft approach. Technical report, Bell Labs, 2003.

[116] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *Software Tools for Technology Transfer*, 2(4) :366–381, 2000.

[117] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[118] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5) :279–295, 1997.

[119] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.

[120] Kenneth L. McMillan. The smv system. Technical report, Carnegie Mellon University, 1992.

[121] Christophe Pajault. Extending quasar with dynamic tasks computation. Technical report, Conservatoire National des Arts et Métiers, Paris, 2005.

[122] Willem Visser and Peter C. Mehlitz. Model checking programs with java pathfinder. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, page 27. Springer-Verlag, 2005.

Mémoire partagée

[123] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96 : Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM Press.

Systèmes répartis

[124] Jean Bacon. *Concurrent Systems : Operating Systems, Database and Distributed Systems : An Integrated Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[125] E. W. Dijkstra. Shmuel safra's version of termination detection. Technical Report EWD-Note 998, University of Texas at Austin, 1987.

[126] E W Dijkstra, W H J Feijen, and A J M van Gasteren. Derivation of a termination detection algorithm for distributed computations. In *Proc. of the NATO Advanced Study Institute on Control flow and data flow : concepts of distributed programming*, pages 507–512, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[127] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3) :37–45, 1986.

[128] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3) :161–175, 1987.

[129] Jayadev Misra. Detecting termination of distributed computations using markers. In *PODC '83 : Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 290–294, New York, NY, USA, 1983. ACM.

[130] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2001.

[131] Dieter Zöbel. The deadlock problem : a classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17(4) :6–15, 1983.

Ada

[132] Pierre Breguet and Luigi Zaffalon. *Programmation concurrente et temps réel avec ADA 95*. Presses polytechniques et universitaires romandes, 2003.

[133] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 1995.

[134] Laura K. Dillon. A visual execution model for ada tasking. *ACM Trans. Softw. Eng. Methodol.*, 2(4) :311–345, 1993.

[135] Laura K. Dillon. Task dependence and termination in Ada. *ACM Transactions on Software Engineering and Methodology*, 6(1) :80–110, 1997.

[136] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652 : 1995(E)*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

Divers

[137] YAHODA Verification Tools Database. `http://anna.fi.muni.cz/yahoda`.

[138] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, 2002.

[139] Patrice Godefroid. The soundness of bugs is what matters. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[140] Marko Mäkelä. Applying compiler techniques to reachability analysis of high-level models. In *Proceedings of the Workshop on Concurrency, Specification and Programming*, pages 129–142. Humboldt-Universität zu Berlin, 2000.

[141] Guy Pujolle. *Les Réseaux*. Eyrolles, 5e edition, 2005.

[142] Sartaj Sahni and Ellis Horowitz. *Fundamentals of Data Structures*. Computer Science Press, 1976.

[143] Andrew S. Tanenbaum. *Réseaux*. Pearson Education, 3e edition, 2003.

[144] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I : Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

Annexes

Annexe

A

Modèles

Sommaire

Nous présentons ici les modèles utilisés lors de cette thèse. Nous effectuons à chaque fois une rapide description du modèle et présentons également le réseau de Petri correspondant lorsque celui-ci est suffisamment lisible.

A.1 La base de données distribuée (*Distributed Database Manager*)

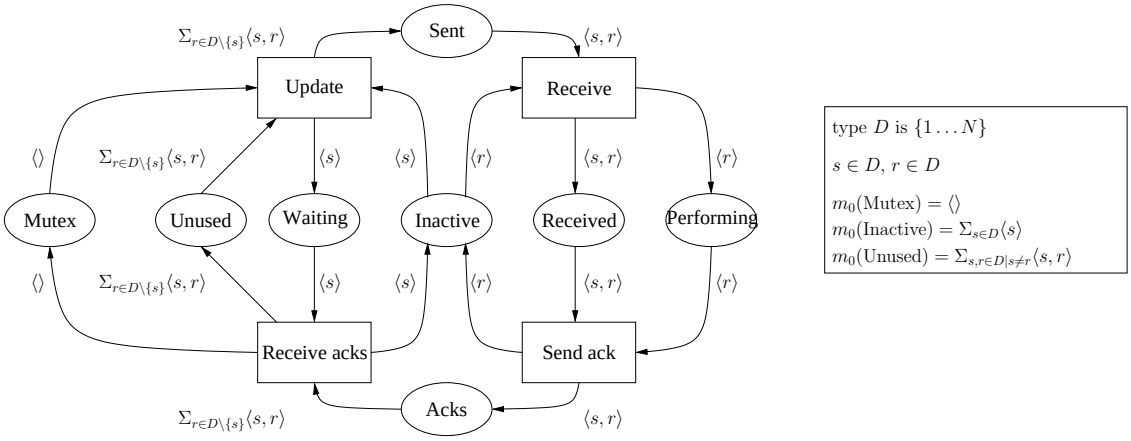


FIG. A.1 – Le modèle de la base de données distribuée

Nous considérons dans ce système un ensemble de N *managers* de base de donné qui communiquent afin de maintenir la cohérence du réplicat de la base. Ce modèle est classique des réseaux de Petri. Il fut initiallement présenté par Genrich puis repris par Jensen.

Lorsqu'un *manager* met à jour sa copie locale de la base de donnée, il envoie des requêtes aux autres *managers* pour qu'ils maintiennent également leurs copies locales (transition `Update`). Dès qu'un *manager* reçoit une telle requête (transition `Receive`), il met à jour sa copie. La mise à jour terminée, chaque *manager* envoie un acquittement à l'initiateur (transition `Send ack`). Le processus se termine lorsque l'initiateur a récupéré tous les acquittements (transition `Receive acks`).

Un *manager* peut être dans différents états :
– il peut être inactif (`Inactive`),
– en attente d'aquittements (`Waiting`),
– en train d'effectuer une mise à jour (`Performing`)

Les places `Msgs`, `Received`, `Acks` et `Unused` sont utilisées pour modéliser les canaux de communication entre les différents sites. En tout, $N.(N-1)$ jetons sont présents dans les différentes places du réseau. La correction du protocole est assurée par la place `Mutex` qui modélise un verrou garantissant que deux *managers* ne peuvent mettre à jour la même copie locale concurrement.

A.2 Equilibreur de charge (*Load Balancer*)

Le système modélisé ici représente un système d'équilibrage de charge simple. Nous avons alors deux types de processus : les clients et les serveurs.

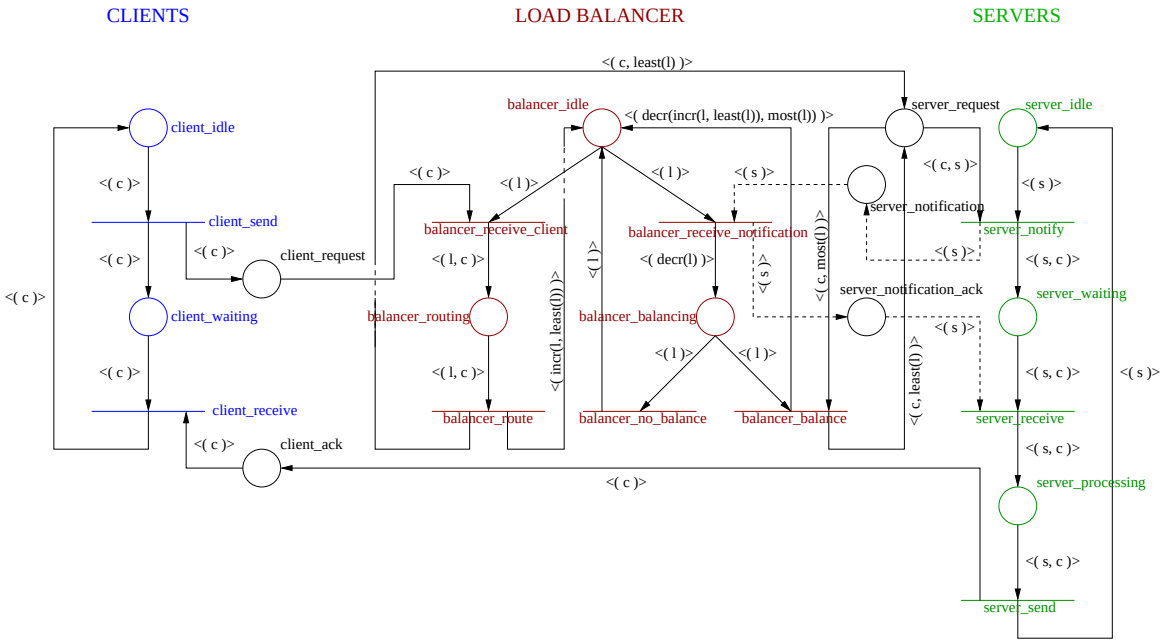


FIG. A.2 – Le modèle de l'équilibreur de charge

A.2.1 Les clients

Nous notons C le nombre de clients. Ils sont numérotés de 1 à C. Le comportement d'un client est assez simple : lorsqu'un client décide d'envoyer une requête aux serveurs, plutôt que de contacter un serveur directement, il envoie un requête à l'équilibreur de charge. Celui-ci transmettera la requête au serveur adéquat, i.e., le serveur le moins chargé. Une fois sa requête envoyée, le client se met alors en attente de la réponse avant de retourner dans l'état inactif.

A.2.2 Les serveurs

Le nombre de serveurs est noté S. Les serveurs sont numérotés de 1 à S. Les serveurs reçoivent les requêtes des client *via* l'équilibreur de charge. Lorsqu'un serveur accepte un requête, il notifie dans un premier temps l'équilibreur afin que ce dernier puisse rééquilibrer la charge lors de la réception des requêtes suivantes. Le serveur se met ensuite en attente d'un acquittement de l'équilibreur avant de traiter la requête et d'envoyer directement la réponse au client concerner avant de revenir dans l'état inactif.

A.2.3 L'équilibreur de charge

L'équilibreur de charge peut effectuer deux types d'opération. La première consiste à rediriger les reques des clients vers les serveurs les moins chargés. La seconde consiste rééquilibrer les requête lorsqu'un serveur a accepter la requête d'un client. Si les requêtes sont déjà équilibrées, l'équilibreur n'a rien a effectuer et peut retourner dans l'état inactif (transition `balancer_no_balance`).

A.3 L'allocateur de ressources (*Allocator*)

Ce réseau modélise un système d'allocation de ressources.

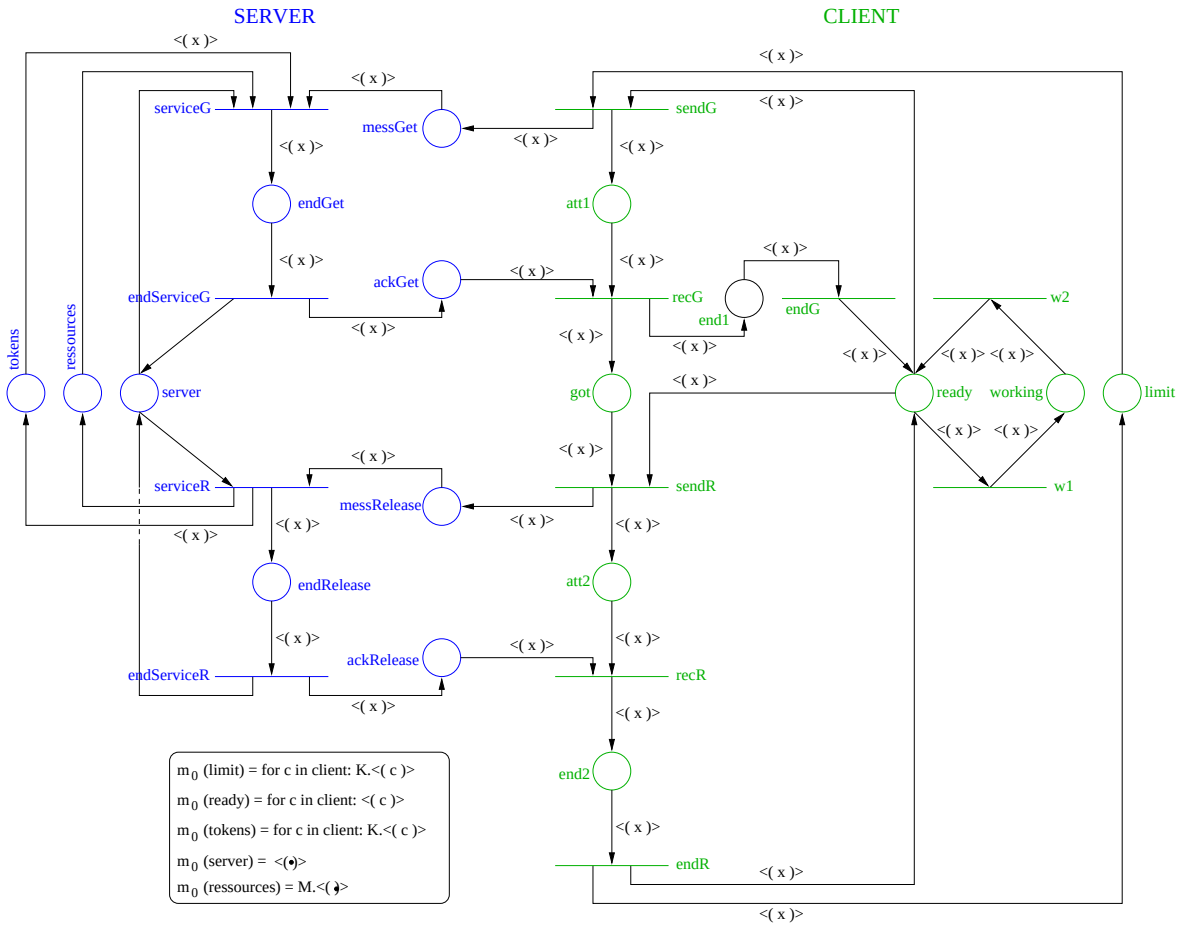
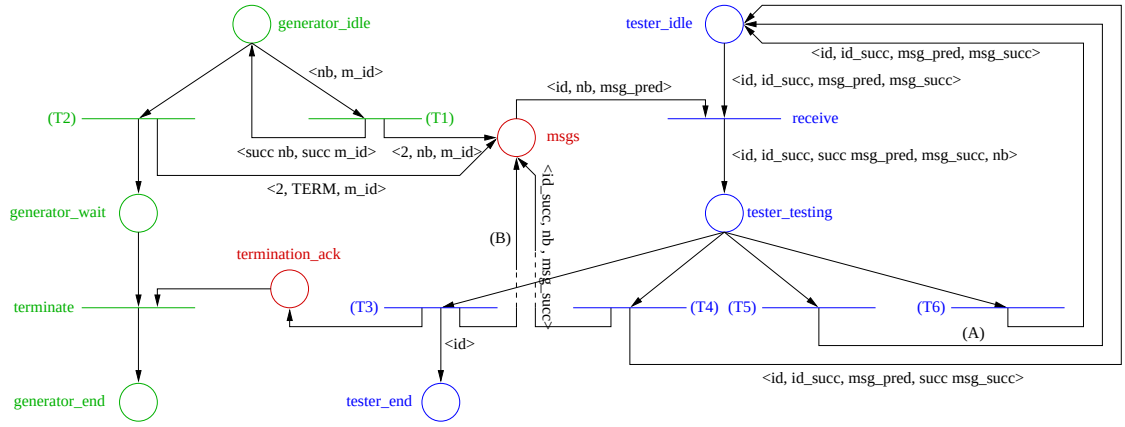


FIG. A.3 – Le modèle de l'allocateur de ressources

M ressources sont disponibles, et chaque client peut obtenir un maximum de K ressources. Initialement, un client peut : soit demander l'acquisition d'une nouvelle ressource (transition `sendG`), soit travailler localement
Une fois la ressource obtenue (transition `recG`), deux choix sont alors possibles :
– recommencer le processus en travaillant et/ou récupérant une autre ressource si la limite des ressources n'est pas atteinte
– libérer une ressource (transition `sendR`) avant de retourner dans l'état initial.

A.4 Crible d'Eratosthene (*Sieve of Eratosthene*)

Ce modèle représente une solution au crible d'Eratosthene pour la génération des nombres premiers de 1 à N. Cette version correspond à une modélisation possible du programme présenté à la figure 7.22 : une solution dynamique et hautement parallèle.



(A) <id, id_succ, msg_pred, msg_succ> + <nb, NO_SUCC, FIRST_MSG, FIRST_MSG>
(B) if (succ_id != NO_SUCC): <id, id_succ, msg_pred, msg_succ, nb>

(T1) generate [guard: nb <= N]
(T2) send_termination [guard: nb>N]
(T3) tester−terminate [guard: nb=TERMINATION]
(T4) not_div_succ [guard: (nb!=TERMINATION) and (nb mod id != 0) and (id_succ != NO_SUCC)]
(T5) not_div_no_succ [guard: (nb!=TERMINATION) and (nb mod id !=0) and (id_succ = NO_SUCC)]
(T6) divide [guard: nb!=TERMINATION and nb mod id=0]

FIG. A.4 – Le crible d'Eratosthene

A.4.1 Le générateur

Le générateur a pour rôle de générer tous les nombres i de 1 à N et de les passer à un premier *tester* qui va servir de premier crible pour tester la primalité de i (transition `generate`). Une fois tous les nombres envoyés, le générateur envoie un message de terminaison au premier *tester* et attend le message d'acquittement de la terminaison pour s'arrêter (transition `terminate`).

A.4.2 Les testers

Les *testers* sont des processus qui vont garder chacun un nombre premier comme crible. Lorsqu'un tester reçoit un message, quatre cas sont possibles :

- transition `divide` : le message reçu est un nombre à tester divisible par le nombre premier stocké par le *tester* (en fait, son identifiant `id`). Dans ce cas là, le nombre n'est pas premier et n'est donc pas conservé.
- transition `not_div_succ` : le message reçu est un nombre à tester *non* divisible par le nombre premier stocké par le *tester*. Dans ce cas là, le nombre est peut être premier. Le *tester* le passe alors à son successeur pour continuer le test.
- transition `not_div_no_succ` : le message reçu est un nombre à tester *non* divisible par le nombre premier stocké par le *tester* et le *tester* ne possède pas de successeur : le nombre est donc premier. Le *tester* crée alors un nouveau processus *tester* à qui il donne comme identifiant la valeur du nombre premier. Ce nouveau processus est alors un nouveau crible

– transition `tester_terminate` : le message reçu est un message de terminaison, si le *tester* est le dernier du crible il acquitte le générateur, sinon, il transmet le message de terminaison à son successeur.

Ce modèle est donc assimilable à un *pipeline* de processus dont chaque étage correspond à une entrée du crible et un nombre premier.
Les conditions pour entrer dans un des 4 choix possibles sont exprimées dans les gardes des transitions correspondantes.

A.5 Election du leader

Ce modèle présente le protocole d'élection de Chang et Roberts sur un anneau unidirectionnel.
Le modèle est très petit (4 places et 7 transitions) mais est assez difficile à lire c'est pourquoi nous n'en donnons pas la représentation graphique.

A.6 Multiprocesseur (*Multiprocessor*)

Ce modèle (présenté dans [29]) décrit l'architecture d'un multiprocesseur.

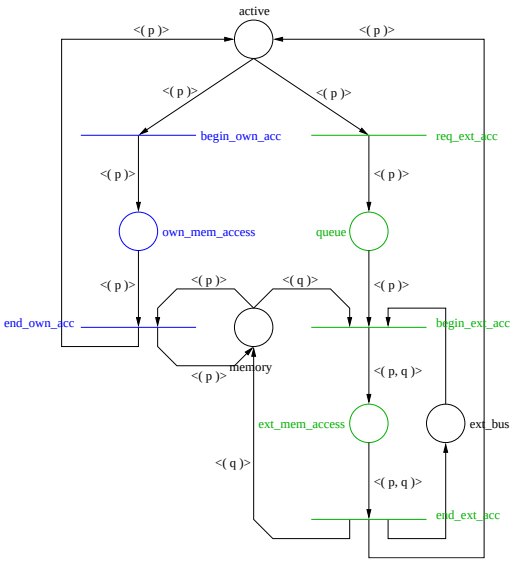


FIG. A.5 – Le modèle du multiprocesseur

Il consiste en un ensemble de processeurs. Chaque processeur peut accéder sa mémoire locale en utilisant le bus local (transition `begin_own_acc`), ou la mémoire associée à un autre processeur en utilisant le bus externe unique (place `ext_bus`) partagé par tous les processeurs (transition `req_ext_acc`).

A.7 Anneau à jeton (*Slotted Ring Protocol*)

Ce modèle est décrit et présenté sous forme graphique dans [24].

A.8 Dîner des philosophes (*Dining philosophers*)

Ce modèle décrit une spécification du modèle des philosophes. Cette solution est garantie sans famine ni interblocage.

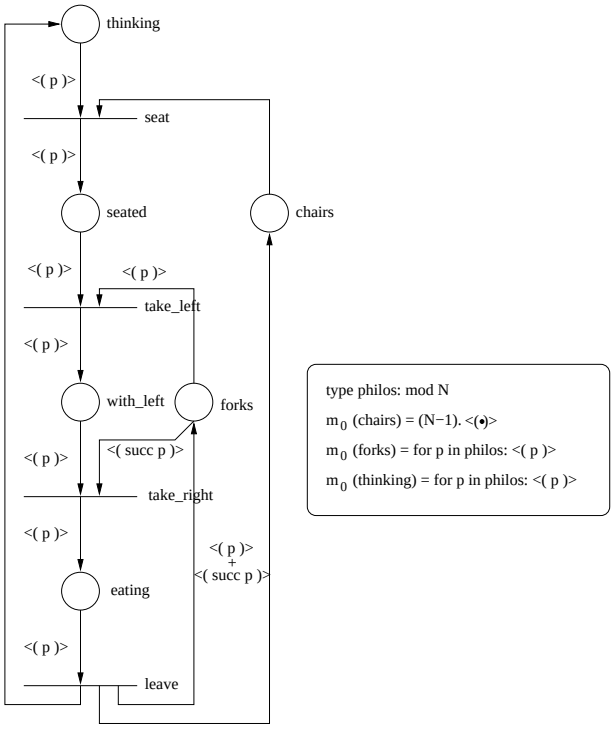


FIG. A.6 – Le modèle du dîner des philosophes

Le système comprend N philosophes. L'interblocage est évité par l'utilisation des $N-1$ chaises. Chaque philosophe commence par prendre une chaise (transition `seat`). Puis, il prend successivement sa fourchette gauche (transition `take_left`) puis sa fourchette gauche (transition `take_right`) qui est celle de son voisin (jeton `succ p`). Une fois ses deux fourchettes acquise il peut alors manger (place `eating`) avant de rendre ses deux fourchettes et quitter la table en rendant sa chaise (transition `leave`).

A.9 Peterson

Ce système décrit l'algorithme d'exclusion mutuelle de Peterson. Le réseau modélisant ce système est assez complexe c'est pourquoi nous ne le présentons pas ici. L'algorithme est rappelé ci-dessous :

```
flag[0] = flag[1] = turn = 0

P0: flag[0] = 1                          P1: flag[1] = 1
    turn = 1                                 turn = 0
    while( flag[1] && turn == 1 );           while( flag[0] && turn == 0 );
            // do nothing                            // do nothing
    // critical section                      // critical section
    ...                                      ...
    // end of critical section               // end of critical section
    flag[0] = 0                              flag[1] = 0
```

A.10 Système de communication pair-à-pair (*Peer-to-peer*)

Ce réseau décrit un exemple de système pair-à-pair simple utilisant un serveur pour trouver un autre pair avec qui communiquer.

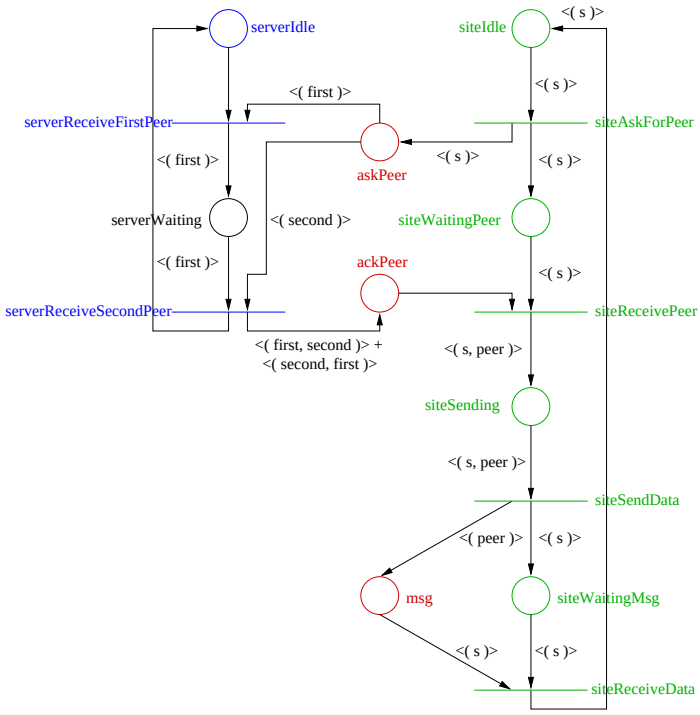


FIG. A.7 – Le système de communication pair-à-pair.

A.10.1 Les sites

Pour obtenir un rendez-vous avec un autre site, un site envoie son identifiant au serveur (transition `siteAskForPeer`) puis attend la réponse du serveur (transition `siteReceivePeer`). Cette réponse est composée de l'identifiant du site appelant et du site avec lequel il pourra communiquer. Le site peut alors envoyer un message à son site partenaire (transition `siteSendData`) et recevoir un message du partenaire (transition `siteReceiveData`).

A.10.2 Le serveur

Dans l'état inactif, le serveur attend une requête de communication d'un site (transition `serverReceiveFirstPeer`). Une fois la requête reçue, il se met en attente d'une seconde requête (transition `serverReceiveSecondPeer`). Une fois cette seconde requête reçue, il acquitte les deux sites et les notifie de leur partenariat (jetons déposés dans la place `ackPeer`).

A.11 Eisenberg & McGuire

Le modèle Eisenberg & McGuire décrit la solution au problème de l'exclusion mutuelle pour N processus décrite par Eisenberg & McGuire dont une description détaillée peut être trouvée dans [124].

Le réseau de Petri associé étant assez grand, nous ne présentons ici que l'algorithme (donné à la figure A.8). Le concept est assez simple.

La variable `turn` et le *flag* `status` sont utilisés de la même manière que dans l'algorithme de Dekker pour le cas à 2 processus. Les *flags* peuvent prendre trois valeurs distinctes :
– `WAITING` lorsque le processus est en attente de la ressource
– `ACTIVE` lorsque le processus est en section critique
– `IDLE` sinon
La priorité des processus en maintenue de façon circulaire en commençant par le processus détenant le *turn*. Chaque processus commence par scanner tous les processus à partir de celui avec le *turn* jusqu'à lui-même.
Lorsque tous les processus ont été détectés comme inactifs, le processus passe provisoirement à l'état `ACTIVE`. Cependant, il est encore possible qu'un autre processus ayant démarré un scan après et se situant avant le processus courant soit également dans cet état. Il est alors nécessaire d'effectuer une nouvelle vérification pour être sûr qu'aucun autre processus ne soit actif.

Le processus peut alors entre en section critique avant de retourner dans l'état `IDLE`.

```
---  variables  ---
constant int N = 4;
shared enum states {IDLE, WAITING, ACTIVE} flags[n -1];
shared int turn;
int i; //  not shared

--  initialization  --
...
turn = 0;
...
for (i=0; i<n; i++)
{
  flags[i] = IDLE;
}

---  code for a process p (p in [0..N-1])  ---
01 loop {
02   repeat {
03     // announce that we need the resource
04     flags[p] = WAITING;
05     //  scan processes from the one with the turn up to ourselves.
06     //  repeat if necessary until the scan finds all processes idle
07     i = turn;
08     while(i != p) {
09       if(flag[i] != IDLE)
10         i = turn;
11       else
12         i = i + 1 mod N;
13     }
14     //  now tentatively claim the resource
15     flags[p] = ACTIVE;
16     //  find the first active process besides ourselves, if any
17     i = 0;
18     while (i < n and (i == p or flags[i] != ACTIVE)) {
19       i = i + 1;
20     }
21     //  if there were no other active processes, AND if we have the
22     //  turn or else whoever has it is idle, then proceed.
23     //  Otherwise, repeat the whole sequence.
24   }
25   until (i >= n and (turn == p or flags[turn] == IDLE));
26   //  claim the turn and proceed
27   turn = p;
28   //  critical section
29   ...
30   //  find a process which is not IDLE
31   //  (if there are no others, we will find ourselves)
32   i = turn + 1 mod n;
33   while (flags[i] = IDLE) {
34     i = i + 1 mod n;
35   }
36   //  give the turn to someone that needs it, or keep it
37   turn = i;
38   //  we're finished now
39   flag[p] = IDLE;
40 }
```

FIG. A.8 – L'agorithme de Eisenberg & McGuire

Annexe

B

Résultats expérimentaux

Sommaire

Dans cette annexe, nous présentons les résultats expérimentaux obtenus par Cyclades. Ces résultats nous ont permis de créer les graphes et courbes présentés dans cette thèse.

B.1 Model checking réparti

Nous présentons ici les résultats expérimentaux associés aux différents graphes et courbes présentés dans cette thèse.

B.1.1 Evaluation du partitionnement (p.238 à p.248)

Cette première sous-section présente les résultats obtenus pour l'évaluation des différentes stratégies de partitionnement pour le model checking réparti.

Pour chaque modèle, nous rappelons le nombre de places ($\mathcal{P}$) et transitions($\mathcal{T}$) du réseau initial puis du réseau réduit après une première analyse statique ($\mathcal{P}_r$ et $\mathcal{T}_r$).

Pour chaque modèle, nous donnons :
- le nombre de nœuds utilisés pour l'exploration (n)
- l'écart-type obtenu comparé à une répartition optimale (lorsque le nombre d'états est également réparti sur chacun des nœuds) noté (σ)
- le nombre d'état stockés sur la partition de plus petite et de plus grand taille ($\mathcal{P}_{min}$ et$\mathcal{P}_{max}$)
- le degré de localité obtenu évalué en pourcentage d'états s' successeurs de s explorés pour lesquels la partition de s' est la même que la partition de s (δ).
- le nombre de communications totales ($\mathcal{M}$)
- le nombre d'états envoyés ($\mathcal{M}_s$)

B.1.2 Evaluation du Δ-marking (p.249 et p.250)

Pour chaque modèle, nous donnons ici le nombre de marquages stockés symboliquement (Δ-marquages) obtenus avec les différentes stratégies de partitionnement.

B.1.3 Evaluations du temps d'exploration (p.250)

Pour chaque modèle, nous donnons ici le temps d'exécution (*time*) exprimé en secondes ainsi que le *ratio*, c'est-à-dire le gain de temps obtenu donné par la formule :

$$\frac{temps\ réparti}{temps\ séquentiel}$$

.

B.2 Model checking multithreadé

Les tests ont été effectués sur un Xéon équipé de deux processeurs quadri-core. Pour chaque modèle analysé, les temps d'exécution ainsi que le gain de temps obtenu sont reportés.

Pour chaque modèle, les simulations ont été exécutées de 2 façons différentes :

1. Δ-*marking :* exploration en utilisant la méthode des Δ-*markings* avec une profondeur de 20.
2. Δ-*marking + state collapse :* exploration avec les deux méthodes (Δ-*markings* et *state collapsing*) couplées.

Pour chaque modèle, les valeurs suivantes sont reportées :
– $|\mathcal{P}|$: le nombre de places dans le réseau initial
– $|\mathcal{P}_r|$: le nombre de places dans le réseau réduit à l'aide des réductions structurelles
– $|\mathcal{T}|$: le nombre de transitions dans le réseau initial
– $|\mathcal{T}_r|$: le nombre de transitions dans le réseau
– *états* : le nombre d'états explorés
– *arcs :* le nombre d'arcs explorés
– Δ : le nombre d'états stocké de manière symbolique pour la technique des Δ-*markings* est utilisée
– *explicit* : le nombre d'états stocké de manière explicite pour la technique des Δ-*markings* est utilisée

Pour chaque type d'exploration, le temps est reporté (t) ainsi que le *ratio* obtenu comparé au temps séquentiel pour chaque valeur th représentant le nombre de *threads* utilisés pour l'exploration.

		Partitionnement uniforme					
	n	σ	$\mathcal{P}_{min}$	p_{max}	δ	$\mathcal{M}$	$\mathcal{M}_s$
Database Manager							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 7/7 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 4/4 états : 22 320 523 arcs : 193 444 552	2	10	11 160 254	11 160 269	<1	352 481	193 444 525
	3	10 206	7 429 968	7 450 380	50	208 666	96 722 277
	4	7	5 580 120	5 580 135	<1	381 699	193 444 539
	5	2 306	4 461 744	4 467 022	<1	384 666	193 444 553
	6	4 564	3 714 984	3 725 190	<1	387 889	193 444 539
	7	479	3 188 458	3 189 733	<1	366 312	193 444 553
	8	4 724	2 781 324	2 798 810	<1	419 042	193 444 539
	9	15 939	2 458 638	2 500 680	<1	408 213	193 444 539
	10	43 518	2 177 448	2 287 558	<1	352 277	193 444 553
Eisenberg & McGuire							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 22/17 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 25/20 états : 33 055 447 arcs : 164 950 145	2	968	16 527 039	16 528 408	23	152 390	125 941 381
	3	11 369	11 008 910	11 031 050	13	205 473	142 171 607
	4	669	8 262 897	8 264 427	12	215 976	144 023 358
	5	48 960	6 549 684	6 667 008	8	272 932	150 502 457
	6	62 231	5 424 365	5 584 545	10	289 568	147 486 157
	7	15 155	4 701 357	4 739 470	17	253 510	134 962 725
	8	1 841	4 129 055	4 133 973	8	287 169	150 502 457
	9	8 652	3 663 490	3 690 033	8	277 057	150 502 457
	10	24 955	3 269 397	3 340 329	8	276 35	150 502 457

Tab. B.1 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (1)

		Partitionnement uniforme					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Eratosthene							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 12 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7 états : 24 135 064 arcs : 135 712 001	2	2	12 067 530	12 067 534	19	505 137	109 862 583
	3	9 195	8 035 399	8 053 720	0	594 034	135 010 686
	4	1 673	6 031 763	6 035 771	8	438 685	124 689 947
	5	70	4 826 919	4 827 073	0	652 840	135 712 001
	6	4 112	4 017 694	4 026 878	0	793 433	135 712 002
	7	4 845	3 441 536	3 454 009	0	596 810	135 712 002
	8	925	3 015 732	3 017 988	0	419 657	135 712 002
	9	2 805	2 677 710	2 685 196	0	640 612	135 712 002
	10	4 016	2 407 860	2 419 205	0	425 233	135 712 002
Dining philosophers							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 12 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7 états : 42 981 185 arcs : 610 909 312	2	1	21 490 592	21 490 593	25	3 728 400	456 281 441
	3	488 536	14 212 059	14 887 223	37	3 202 537	381 851 089
	4	1 675	10 744 285	10 745 301	0	5 954 912	606 977 153
	5	1 327	8 594 640	8 597 834	0	5 639 784	610 909 313
	6	218 480	6 914 496	7 402 736	<1	924 527	610 909 313
	7	45 548	6 081 442	6 197 428	<1	1 036 510	608 812 161
	8	18 570	5 349 600	5 395 569	<1	914 867	610 909 313
	9	141 287	4 598 949	4 943 884	<1	993 226	610 909 313
	10	77 426	4 166 080	4 431 144	<1	940 198	610 909 313

Tab. B.2 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (2)

		Partitionnement uniforme					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Leader election protocol							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 4/4 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7/7 états : 10 475 430 arcs : 117 722 093	2	1 793 577	3 969 464	6 505 966	92	20 156	9 063 626
	3	0	3 491 810	3 491 810	2	218 417	114 230 284
	4	732 225	1 984 732	3 254 275	31	135 843	79 589 926
	5	1 476	2 093 390	2 096 933	0	252 719	117 689 326
	6	463 099	1 323 154	2 168 656	0	291 537	117 689 326
	7	10 144	1 483 214	1 509 211	0	208 978	117 689 326
	8	338 954	992 275	1 627 160	0	248 305	117 689 326
	9	40 867	1 110 888	1 218 027	0	207 382	117 689 326
	10	267 372	792 290	1 301 809	0	271 441	117 689 326
Peer-to-peer							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 9/8 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 6/5 états : 5 213 808 arcs : 35 986 356	2	219 525	2 451 676	2 762 132	62	16 740	13 384 441
	3	46 616	1 703 916	1 791 072	8	65 300	33 003 289
	4	89 838	1 218 988	1 384 476	18	38 200	29 294 137
	5	24 746	1 012 656	1 077 432	28	37 799	25 584 985
	6	60 691	809 820	941 904	<1	59 976	35 986 357
	7	44 778	675 544	799 707	<1	48 526	35 986 357
	8	46 833	570 898	697 722	18	43 130	29 294 137
	9	23 357	536 238	615 834	<1	45 984	35 986 357
	10	34 901	477 537	574 455	<1	46 637	35 986 357

TAB. B.3 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (3)

		Partitionnement basé sur la détection des points de synchronisation					
	n	σ	$\mathcal{P}_{min}$	p_{max}	δ	$\mathcal{M}$	$\mathcal{M}_s$
Database Manager							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 7/7 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 4/4 états : 22 320 523 arcs : 193 444 552	2	1	11 160 261	11 160 262	46	188 291	104 162 451
	3	56 056	7 387 687	7 499 223	32	258 481	129 671 623
	4	11 852	5 568 319	5 591 943	23	293 794	148 803 501
	5	15 643	4 448 898	4 486 023	19	295 583	155 180 795
	6	26 738	3 684 582	3 760 857	15	302 546	163 683 851
	7	2 730	3 187 614	3 194 839	14	302 093	165 809 617
	8	14 273	2 774 598	2 813 970	10	313 970	172 186 909
	9	30 261	2 411 290	2 501 928	9	319 002	174 312 673
	10	12 582	2 212 483	2 252 575	8	324 423	176 438 437
Eisenberg & McGuire							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 22/17 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 25/20 états : 33 055 447 arcs : 164 950 145	2	3 102 322	14 334 050	18 721 397	68	72 595	49 869 084
	3	1 841 742	9 665 864	13 116 000	57	107 847	68 877 896
	4	1 625 019	7 112 362	10 606 469	50	154 463	79 731 010
	5	1 234 909	5 890 984	8 808 787	45	232 665	88 367 317
	6	1 440 748	4 485 908	8 394 534	46	272 754	85 007 784
	7	1 004 574	4 005 579	6 943 908	41	240 776	94 073 351
	8	962 091	3 512 575	6 418 930	38	775 393	98 846 518
	9	793 429	3 404 005	5 694 111	36	802 298	102 405 094
	10	875 650	2 550 964	5 674 619	36	460 242	101 588 779

Tab. B.4 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (4)

		Partitionnement basé sur la détection des points de synchronisation					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Eratosthene							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 12 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7 états : 24 135 064 arcs : 135 712 001	2	2	12 067 530	12 067 534	19	505 137	109 862 583
	3	9 195	8 035 399	8 053 720	0	594 034	135 010 686
	4	1 673	6 031 763	6 035 771	8	438 685	124 689 947
	5	70	4 826 919	4 827 073	0	652 840	135 712 001
	6	4 112	4 017 694	4 026 878	0	793 433	135 712 002
	7	4 845	3 441 536	3 454 009	0	596 810	135 712 002
	8	925	3 015 732	3 017 988	0	419 657	135 712 002
	9	2 805	2 677 710	2 685 196	0	640 612	135 712 002
	10	4 016	2 407 860	2 419 205	0	425 233	135 712 002
Dining philosophers							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 12 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7 états : 42 981 185 arcs : 610 909 312	2	11 775	21 482 267	21 498 918	47	645 781	323 786 354
	3	21 124	14 314 123	14 353 154	54	546 789	281 023 214
	4	23 157	10 731 159	10 761 153	34	756 413	397 095 101
	5	28 146	8 554 698	8 612 156	22	894 573	476 541 254
	6	12 156	7 158 153	7 165 543	27	887 642	445 967 423
	7	18 165	6 122 345	6 153 321	21	896 532	482 619 532
	8	21 147	5 332 157	5 391 154	18	932 143	500 946 143
	9	25 315	4 753 156	4 784 216	16	961 453	513 163 824
	10	26 154	4 274 486	4 410 056	19	976 513	494 836 643

TAB. B.5 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (5)

		Partitionnement basé sur la détection des points de synchronisation					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Leader election protocol							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 6/5 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 5/4 états : 10 475 430 arcs : 117 722 093	2	17 041	5 225 665	5 249 765	68	58 488	36 648 733
	3	34	3 491 771	3 491 836	22	145 547	91 751 886
	4	6 957	2 612 747	2 624 919	33	125 120	78 582 091
	5	24	2 095 059	2 095 117	12	164 623	102 687 795
	6	4 401	1 741 759	1 750 064	16	155 808	98 050 070
	7	145	1 496 352	1 496 728	13	163 488	101 970 394
	8	3 222	1 306 254	1 312 565	15	163 315	99 723 165
	9	170	1 163 683	1 164 207	6	207 101	110 038 420
	10	2 544	1 044 996	1 050 145	10	210 693	105 790 739
Peer-to-peer							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 9/8 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 6/5 états : 5 213 808 arcs : 35 986 356	2	170 452	2 486 376	2 727 432	68	13 748	11 153 701
	3	336 430	1 543 698	2 126 412	62	17 984	13 384 441
	4	243 473	1 092 192	1 635 240	55	21 148	15 615 181
	5	192 215	956 800	1 386 608	49	24 565	17 845 921
	6	188 585	689 706	1 214 706	49	25 294	17 845 921
	7	149 286	630 144	1 070 983	49	24 818	17 845 921
	8	183 983	545 564	1 089 676	49	26 808	17 845 921
	9	185 680	514 566	1 074 282	49	38 269	17 845 921
	10	127 209	461 905	879 812	43	52 156	20 076 661

TAB. B.6 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (6)

	n	Partitionnement basé sur la détection de cycles σ	$\mathcal{P}_{min}$	p_{max}	δ	$\mathcal{M}$	$\mathcal{M}_s$
Database Manager							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 7/7 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 4/4 états : 22 320 523 arcs : 193 444 552	2	7 891 490	5 580 135	16 740 388	33	185 018	96 722 277
	3	32 805	7 407 369	7 472 979	31	295 998	131 265 946
	4	3 221 687	2 790 046	8 372 063	17	325 789	141 894 769
	5	652	4 463 622	4 465 192	17	311 106	160 495 208
	6	2 037 658	1 855 366	5 610 027	11	369 830	158 900 883
	7	142	3 188 323	3 188 700	14	331 413	165 809 617
	8	1 491 351	1 394 874	4 186 908	8	415 436	167 403 940
	9	9 471	2 468 970	2 491 083	8	332 632	176 969 879
	10	1 176 394	1 115 454	3 348 755	5	327 309	176 969 880
Dining philosophers							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 12 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7 états : 42 981 185 arcs : 610 909 312	2	94 865	21 423 513	21 557 672	67	487 214	256 647 213
	3	51 314	14 321 246	14 331 279	72	442 274	231 409 847
	4	20 754	10 721 624	10 772 249	53	535 565	286 388 317
	5	48 207	8 574 678	8 682 473	43	661 924	343 587 337
	6	47 858	7 117 250	7 235 559	49	586 570	310 254 010
	7	920	6 139 431	6 141 159	18	932 350	496 363 817
	8	9 607	5 360 812	5 386 237	26	839 682	448 648 815
	9	29 514	4 755 767	4 815 039	21	897 439	477 281 093
	10	22 735	4 286 574	4 341 853	26	841 336	448 616 047

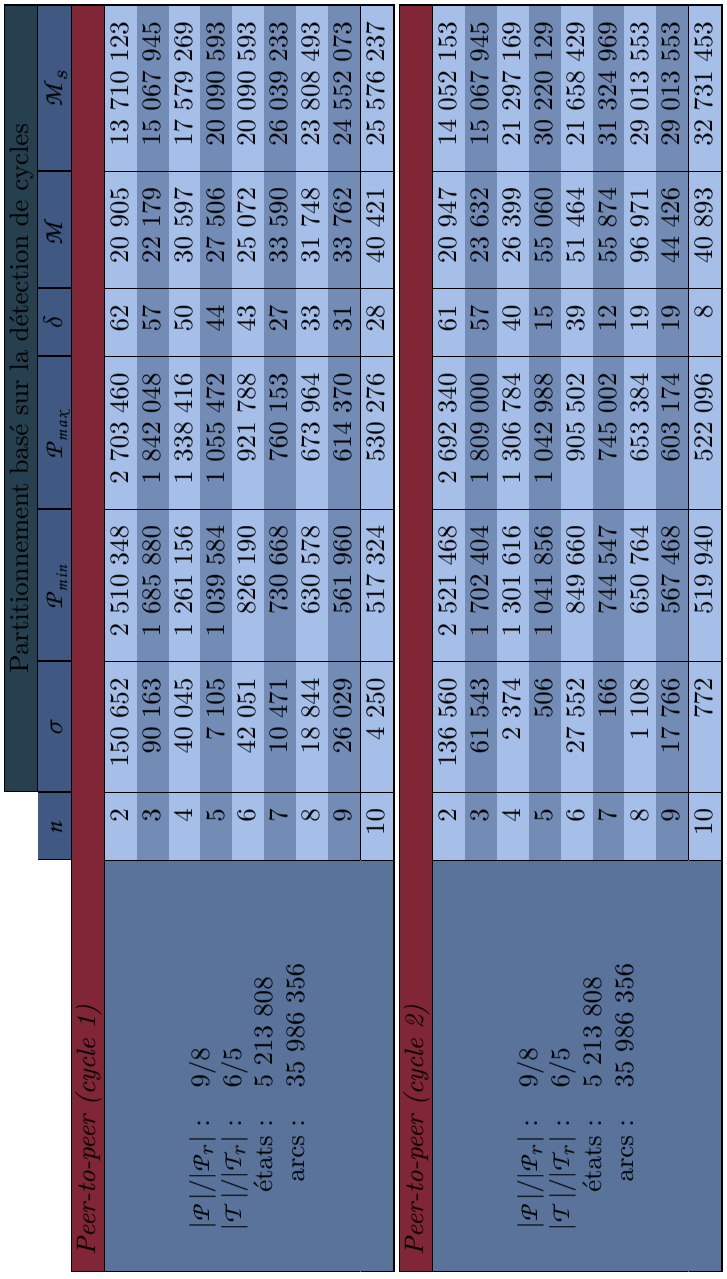
Tab. B.7 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (7)

		Partitionnement basé sur la détection de cycles					
	n	σ	$\mathcal{P}_{min}$	p_{max}	δ	$\mathcal{M}$	$\mathcal{M}_s$
Eisenberg & McGuire (cycle 1)							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 22/17 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 25/20 états : 33 055 447 arcs : 164 950 145	2	632 972	16 080 144	16 975 303	83	33 089	26 525 373
	3	57 643	10 956 929	11 071 195	47	148 203	87 031 224
	4	348 603	7 998 543	8 771 197	63	94 195	59 816 732
	5	117 691	6 530 849	6 819 266	27	262 498	119 157 099
	6	167 251	5 336 105	5 714 905	41	205 686	95 626 547
	7	23 698	4 703 368	4 767 498	23	500 371	125 288 026
	8	161 782	3 988 280	4 403 092	36	287 554	103 751 705
	9	17 427	3 648 638	3 699 656	21	343 586	128 876 982
	10	109 633	3 173 896	3 506 287	24	396 148	124 250 854
Eisenberg & McGuire (cycle 2)							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 22/17 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 25/20 états : 33 055 447 arcs : 164 950 145	2	4 682 862	13 216 440	19 839 007	78	52 443	33 962 482
	3	2 169 411	9 690 501	13 521 955	71	68 388	46 647 132
	4	2 437 265	6 565 462	11 770 521	69	114 002	48 663 802
	5	1 935 046	5 613 107	10 068 395	60	112 956	62 829 111
	6	1 878 082	4 293 844	9 228 111	64	206 021	57 483 462
	7	1 513 053	3 971 852	8 113 831	64	157 095	72 390 956
	8	1 485 957	3 325 489	7 690 168	54	174 640	71 831 012
	9	1 159 494	2 867 631	6 686 429	52	178 789	75 027 139
	10	1 360 920	2 230 961	7 031 854	52	230 356	75 187 474

Tab. B.8 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (8)

		Partitionnement basé sur la détection de cycles					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Eratosthene (cycle 1)							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 8/7 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 8/7 états : 18 879 295 arcs : 104 518 776	2	79	9 439 591	9 439 704	85	45 907	14 972 379
	3	2 732	6 290 265	6 295 717	53	225 364	48 320 964
	4	878	4 719 749	4 720 866	50	234 576	52 145 443
	5	3 506	3 771 195	3 779 851	30	705 938	72 497 305
	6	1 258	3 145 499	3 148 044	47	390 592	55 276 987
	7	697	2 696 181	2 698 204	28	542 228	75 192 844
	8	602	2 358 776	2 360 557	32	846 070	70 875 563
	9	2 483	2 092 293	2 100 584	26	684 641	76 411 909
	10	2 141	1 885 049	1 892 203	28	256 414	74 662 664
Eratosthene (cycle 2)							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 8/7 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 8/7 états : 18 879 295 arcs : 104 518 776	2	11 966	9 431 186	9 448 109	39	275 684	63 757 429
	3	6 490	6 287 556	6 300 239	35	299 281	67 362 090
	4	4 935	4 714 077	4 721 728	20	313 154	82 758 823
	5	3 520	3 771 474	3 779 935	20	730 634	83 287 349
	6	2 904	3 143 686	3 150 144	17	455 508	86 314 366
	7	1 183	2 695 658	2 698 669	11	715 826	92 110 212
	8	2 487	2 356 115	2 364 066	10	298 976	93 491 991
	9	2 055	2 094 779	2 100 633	14	1 302 454	89 606 060
	10	1 678	1 885 696	1 890 082	6	794 386	97 839 313

TAB. B.9 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (9)



		Partitionnement basé sur la détection de cycles					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Peer-to-peer (cycle 1)	2	150 652	2 510 348	2 703 460	62	20 905	13 710 123
	3	90 163	1 685 880	1 842 048	57	22 179	15 067 945
	4	40 045	1 261 156	1 338 416	50	30 597	17 579 269
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 9/8	5	7 105	1 039 584	1 055 472	44	27 506	20 090 593
$\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 6/5	6	42 051	826 190	921 788	43	25 072	20 090 593
états : 5 213 808	7	10 471	730 668	760 153	27	33 590	26 039 233
arcs : 35 986 356	8	18 844	630 578	673 964	33	31 748	23 808 493
	9	26 029	561 960	614 370	31	33 762	24 552 073
	10	4 250	517 324	530 276	28	40 421	25 576 237
Peer-to-peer (cycle 2)	2	136 560	2 521 468	2 692 340	61	20 947	14 052 153
	3	61 543	1 702 404	1 809 000	57	23 632	15 067 945
	4	2 374	1 301 616	1 306 784	40	26 399	21 297 169
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 9/8	5	506	1 041 856	1 042 988	15	55 060	30 220 129
$\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 6/5	6	27 552	849 660	905 502	39	51 464	21 658 429
états : 5 213 808	7	166	744 547	745 002	12	55 874	31 324 969
arcs : 35 986 356	8	1 108	650 764	653 384	19	96 971	29 013 553
	9	17 766	567 468	603 174	19	44 426	29 013 553
	10	772	519 940	522 096	8	40 893	32 731 453

Tab. B.10 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (10)

		Partitionnement basé sur la détection de cycles					
	n	σ	$\mathcal{P}_{min}$	$\mathcal{P}_{max}$	δ	$\mathcal{M}$	$\mathcal{M}_s$
Leader election protocol							
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 4/4 $\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 7/7 états : 10 475 430 arcs : 117 722 093	2	43 679	5 206 829	5 268 601	67	60 063	37 571 273
	3	34	3 491 771	3 491 836	22	191 433	91 751 886
	4	17 832	2 603 329	2 634 337	33	175 013	78 505 933
	5	78	2 094 982	2 095 195	15	201 200	98 910 088
	6	11 279	1 735 413	1 756 410	16	170 302	97 683 440
	7	565	1 495 710	1 497 540	12	244 481	103 359 466
	8	8 255	1 301 545	1 317 274	14	272 838	100 151 941
	9	179	1 163 666	1 164 160	5	216 845	110 894 674
	10	6 523	1 040 849	1 054 226	12	165 948	102 578 957

TAB. B.11 – Evaluation de l'écart-type de la répartition, du degré de localité et du nombre de communications pour les différentes techniques de partitionnement. (11)

	Partitionnement uniforme	Partitionnement synchro	Partitionnement cycles
n	Δ	Δ	Δ
Database Manager (22 320 523 states)			
1	20 702 906	20 702 906	20 702 906
2	14	10 200 500	10 162 264
3	110 670	6 315 321	7 340 182
4	0	5 223 256	3 221 687
5	0	4 716 177	5 028 780
6	0	3 576 125	3 261 185
7	0	3 365 489	3 185 489
8	0	2 668 755	3 696 193
9	0	1 868 276	1 817 464
10	0	1 927 644	2 281 017
Leader election protocol (10 475 430 states)			
1	9 448 739	9 448 739	9 448 739
2	10 012 607	10 310 445	10 310 445
3	3 477 564	9 507 081	9 507 081
4	90 295	8 758 168	8 758 168
5	32 768	6 331 408	6 331 408
6	32 768	8 321 634	8 321 634
7	32 768	5 013 330	5 013 330
8	32 768	5 841 181	5 841 181
9	32 768	4 067 279	4 067 279
10	32 768	6 003 718	6 003 718
Eisenberg & McGuire (33 055 447 states)			
1	29 751 679	29 751 679	29 751 679
2	9 006 137	23 240 789	28 247 323
3	9 195 756	21 329 889	27 854 412
4	9 555 273	20 949 421	28 073 466
5	7 641 713	18 867 493	27 154 456
6	9 425 975	19 013 395	27 531 042
7	13 036 717	19 193 278	27 299 757
8	7 703 185	16 979 074	26 530 083
9	7 884 224	16 705 355	26 427 725
10	8 124 522	16 428 633	26 318 192

Tab. B.12 – Nombre de Δ-*markings* en réparti (1)

	Partitionnement uniforme	Partitionnement synchro	Partitionnement cycles
n	Δ	Δ	Δ
Eratosthene (24 135 064 states)			
1	16 991 368	16 991 368	16 991 368
2	6 819 107	16 991 368	13 012 090
3	701 316	701 316	11 698 124
4	7 426 584	7 426 584	10 570 463
5	1	1	6 004 706
6	0	0	9 790 197
7	0	0	4 749 601
8	0	0	6 191 680
9	0	0	4 000 731
10	0	0	5 201 641
Dining Philosophers (42 981 185 states)			
1	38 683 120	38 683 120	38 683 120
2	16	35 124 657	38 462 434
3	41 798 021	38 451 236	39 246 140
4	44 814	33 215 131	36 340 097
5	0	29 984 513	37 191 384
6	0	30 152 464	34 322 501
7	161 388	29 421 531	34 338 856
8	0	28 542 131	29 752 938
9	0	28 426 421	29 747 087
10	0	28 475 612	29 798 857

Tab. B.13 – Nombre de Δ-*markings* en réparti (2)

	Eisenberg & McGuire		Sieve of Eratosthene		Peer-to-peer		Leader election protocol	
n	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*
1	4 257	–	2 882	–	433	–	2 912	–
2	2 192	1.94	1 195	2.41	343	1.26	1 854	1.57
3	1 287	3.31	853	3.38	211	2.05	1 096	2.66
4	1 069	3.98	625	4.61	165	2.62	691	4.21
5	875	4.87	547	5.27	117	3.7	565	5.15
6	766	5.56	412	6.99	104	4.16	482	6.04
7	671	6.34	389	7.41	98	4.42	409	7.12
8	622	6.84	330	8.73	85	5.1	359	8.11
9	515	8.27	307	9.39	81	5.35	305	9.55
10	464	9.17	286	10.08	77	5.62	264	11

Tab. B.14 – Evaluation du temps d'exploration.

		Δ-*markings*		Δ-*marking* + *state collapse*	
	th	*t*	*ratio*	*t*	*ratio*
Allocator					
	1	48571	1	51246	1
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 17/12	2	27675	1.72	29794	1.8
$\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 12/7	3	18760	2.5	20498	2.71
états : 107 891 127	4	14627	3.13	18909	3.19
arcs : 861 313 422	5	11833	3.78	13380	3.83
Δ : 102 516 261	6	10213	4.42	11515	4.45
explicit : 5 374 866	7	8942	4.84	10544	4.86
	8	7877	5.76	8896	5.76
Eisenberg & McGuire					
	1	2988	1	3112	1
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 22/17	2	1775	1.68	1691	1.84
$\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 25/20	3	1176	2.02	1461	2.13
états : 33 055 447	4	1042	2.28	1341	2.32
arcs : 164 950 145	5	1205	2.48	1230	2.53
Δ : 31 403 771	6	1158	2.58	1202	2.59
explicit : 1 651 676	7	1136	2.63	1183	2.63
	8	1128	2.65	1174	2.65
Dining philosophers					
	1	647	1	661	1
$\lvert\mathcal{P}\rvert/\lvert\mathcal{P}_r\rvert$: 6/5	2	307	2.11	310	2.13
$\lvert\mathcal{T}\rvert/\lvert\mathcal{T}_r\rvert$: 4/3	3	212	3.05	211	3.14
états : 4 766 585	4	165	3.92	166	3.99
arcs : 59 234 672	5	132	4.9	132	5
Δ : 4 528 344	6	110	5.88	103	5.94
explicit : 238 241	7	93	6.96	96	6.92
	8	82	7.89	84	7.9

TAB. B.15 – Evaluations de Cyclades sur une machine parallèle (1)

		simple		*Δ-marking + state collapse*	
	th	*t*	*ratio*	*t*	*ratio*
Peer-to-peer					
	1	7093	1	7214	1
$\|\mathcal{P}\|/\|\mathcal{P}_r\|$: 9/8	2	3645	1.95	3662	1.97
$\|\mathcal{T}\|/\|\mathcal{T}_r\|$: 6/5	3	2605	2.72	2576	2.8
états : 37 948 824	4	2042	3.47	1950	3.7
arcs : 290 258 640	5	1637	4.33	1582	4.56
Δ : 36 031 751	6	1408	5.04	1324	5.45
explicit : 1 917 073	7	1247	5.69	1244	5.8
	8	1119	6.34	1093	6.6
Slotted-Ring					
	1	4006	1	4228	1
$\|\mathcal{P}\|/\|\mathcal{P}_r\|$: 10/6	2	2091	1.92	2157	1.96
$\|\mathcal{T}\|/\|\mathcal{T}_r\|$: 10/7	3	1391	2.88	1438	2.94
états : 24 893 440	4	1068	3.75	1098	3.85
arcs : 207 567 360	5	862	4.65	886	4.77
Δ : 23 650 092	6	721	5.56	742	5.7
explicit : 1 243 348	7	616	6.5	636	6.65
	8	547	7.32	564	7.5
Multiprocessor					
	1	72 103	1	75 243	1
$\|\mathcal{P}\|/\|\mathcal{P}_r\|$: 6/5	2	39 759	1.81	39 602	1.9
$\|\mathcal{T}\|/\|\mathcal{T}_r\|$: 5/4	3	27 880	2.59	27 868	2.7
états : 43 046 721	4	21 979	3.28	22 261	3.38
arcs : 4 362 067 728	5	17 360	4.15	17 498	4.3
Δ : 40 894 438	6	14 945	4.82	15 325	4.91
explicit : 2 152 283	7	13 268	5.43	13 631	5.52
	8	11 993	6.01	12 275	6.13

Tab. B.16 – Evaluations de Cyclades sur une machine parallèle (2)