République Algérienne Démocratique et Populaire
Ministére de l'enseignement Supérieur et de la Recherche Scientifique Faculté des Nouvelles Technologies de l'Information et la Communication Département d'Informatique Fondamentale et ses Applications

Année :
N° d'ordre :
Série :

MÉMOIRE

pour obtenir le diplôme

MASTER en Informatique

Option : Réseaux et Systèmes Distribués

Approche de redistribution d'espace états en vue d'une vérification model checking

Présenté et soutenu publiquement par

Karimou Seyni Ibrahim

le 25 juin 2019

Jury

Pr. Djamel Eddine SAIDOUNI,	Directeur de mémoire
Dr. Bouneb Zine El Abidine,	Co-encadreur
Pr. Salim CHIKHI,	Président
Pr. Djamel Eddine SAIDOUNI,	Examinateur

MASTER

Table des matières

Liste des figures

Liste des tableaux

Remerciements

Dédicaces

Je dédie ce travail à :

etc.

Introduction Générale

Introduction

Depuis la découverte de l'informatique, de nombreuses activités de la vie courante ont été simplifiés. Aujourd'hui, on se sert de l'informatique pour traiter des informations en utilisant des logiciels et des réseaux informatiques. Et pour ce faire, ce système est devenu maniable presque dans tout le domaine de la vie courante (l'alimentation, l'eau, L'entretien de la maison, la construction, le bénévolat, le jardinage, les achats, les loisirs, avionique, ferroviaire, nucléaire, médicale, etc.).

Certains de ces systèmes, tels que les systèmes avioniques, ferroviaires, nucléaires et médicales, sont caractérisés par leurs aspects critiques, complexes et réactifs, et opèrent comme des programmes très concurrents avec des milliers d'entités s'exécutant en parallèle et communiquant sous différentes conditions environnementales. L'analyse de ces systèmes est particulièrement difficile, car un mauvais fonctionnement du système aurait un impact important sur la sécurité ou la vie des personnes, des entreprises ou des biens. En fait, plusieurs catastrophes sont dues à des erreurs de spécification, a titre d'exemple, l'échec du vol inaugural de la fusée Ariane 5 en 1996 a pour origine plusieurs erreurs de spécification du logiciel de commande, et notamment de son système de tolérance aux fautes **?** ; En 2002, un autre exemple dramatique a eu lieu, il s'agit du missile anti-missile Patriote déployé en Arabie Saoudite, que le système de surveillance a refusé de lancer sur un missile ennemi **?** ; Le bug de l'an 2000 est un exemple, causé par une erreur de specification sur les dates **?**.

Les méthodes formelles permettent de parvenir à une preuve, au sens mathématique, du bon fonctionnement du système. Elles se basent sur la description du système avec un langage formel (i.e. non ambigu) muni d'une sémantique précise. Une fois le système décrit, ces méthodes peuvent prouver la correction du modèle en assurant que le comportement du système développé satisfait tous les besoins désirés et les propriétés spécifiées. Cette vérification peut prendre plusieurs formes, dont l'approche basée sur la preuve de théorèmes permet de vérifier de manière paramétrique un système qui est décrit sous la forme d'axiomes **?**, et l'approche basée sur le model checking, permet de confronter un système (sa description opérationnelle) à ses spécifications (les propriétés que l'on attend de lui), de détecter automatiquement des erreurs dans le processus de conception **?**.

Les techniques des méthodes formelles sont automatisables dès lors que l'on possède un modèle formel et des propriétés à vérifier. La puissance de cette méthode réside dans la précision de la réponse obtenue (la propriété est vérifiée ou non), aussi elle fournit un contre-exemple en cas de non satisfiabilité de la propriété dans le modèle (le système ne vérifie pas la propriété spécifiée) permettant ainsi de corriger la source de l'erreur dans le système.

Problématique

Du fait de la taille de l'espace d'états de certains systèmes spécifiés, il est en général impossible de les explorer dans un temps raisonnable, on se heurte au problème de l'explosion combinatoire du nombre d'états à explorer. Ce problème est retardé par la distribution de l'espace d'états sur un réseau de machines connectées afin de tirer profit de la quantité de mémoire et de la puissance de calcul disponibles.

Dans l'implémentation distribuée, le model checking souffre d'un problème majeur engendré par la distribution de l'espace d'états du systèmes spécifié car il est impossible de connaître à l'avance le temps de calcul que prendra l'exploration de l'espace d'états. L'espace d'états est distribué de sorte que des machines pourraient avoir des états trop faciles à calculer. Il est donc difficile de s'assurer que toutes les machines sont utilisées au maximum de leur capacité et donc on ne peut pas arriver à une accélération linéaire.

Notre objectif réside dans la génération distribuée de l'espace d'états, c'est d'établir un compromis entre l'équilibrage de charge entre les différentes machines et la minimisation des coûts de communication. Cependant, l'expérience pratique a montré que chacun de ces deux objectifs pris séparément ne peut suffire pour avoir une bonne distribution de l'espace des états. Plus précisemment, ces deux objectifs sont contradictoires dans le sens où l'amélioration de l'un se fera au détriment de l'autre.

Contributions

Dans l'implémentation distribuée, les solutions proposées dans la littérature travaillent en amont c-à-d avant ou au moment de la génération de l'espace d'états en utilisant des fonctions de partitions. La philosophie de ces méthodes visent à aboutir à la meilleure distribution sans savoir au préalable les applications qui vont exploiter le graphe distribué par la suite. Cependant ces applications sont de nature différentes, et peuvent dans le cas échéant être inconnues. Suite à cela, une nouvelle protocole de redistribution comportementale a été proposée dans la thèse de de Bensetira basée sur des heuristiques en fonction des transitions [BENSETIRA, 2017]. Ainsi nous proposons une nouvelle approche de redistribution en avale de l'espace d'états basée la théorie de jeux et l'analyse des états. L'approche proposée vise à analyser espace d'état tout en extrayant les informations pertinentes sur les états. Ensuite, redistribuer les états suite à leurs pertinences soit migré définitivement soit dupliqués sur d'autres machines, afin de minimiser le nombre de communications entre les machine. Cela est fait par les machines qui cherchent à optimiser leur taux de communications tout en maintenant un bon équilibrage d'états entre les machines à l'aide des seuils prédéfinis pour chaque machine. Ceci permet à une application d'optimiser ses comportements en cumulant ses expériences d'exécution, ainsi grâce à l'utilisation des base de donnée orienté graphe, les prochaines exécutions de l'application seront fait à partir des améliorations gagnés précédemment.

Plan du document

Conformément à ce qui vient d'être exposé dans notre travail de recherche, il se décompose en deux parties :

La première partie est consacrée à la présentation du domaine de notre étude et les principaux concepts utilisés dans ce travail. Elle contient deux chapitres.

Le premier chapitre aborde quelques notions de base sur l'optimisation combinatoire ainsi que de la théorie de graphes. Une introduction sur la distribution de l'espace d'états et pressente les implémentations distribuées de l'espace d'états basé sur le model checking proposées dans la littérature.

Le deuxième chapitre est consacré à la présentation du contexte général de notre travail, rappelle quelques notions sur les réseaux de Petri et les systèmes de transitions

étiquetées ainsi que les structures de Kripke, ensuite, il met l'accent sur la technique du model checking en présentant les concepts de la logique temporelle arborescente, pour présenter par la suite le model cheching distribué.

La seconde partie est dédiée au contribution de notre projet, elle contient un chapitre. Nous présentons l'approche de génération parallèle d'une structure de Kripke distribué à partir d'un réseau de Petri. Ensuite, nous décrivons notre approche de distribution en avale de l'espace d'états guidée par le comportement des applications. Nous finirons avec la présentation d'un système de stockage dédiée aux espace d'états et l'implémentation de cette approche ainsi que l'interprétation des résultats obtenus.

Enfin, la conclusion générale qui englobe l'aboutissement des résultats obtenus ainsi que les perspectives à développer dans l'avenir.

Première partie

Contexte de travail

Chapitre 1

APPROCHE DE PARTITIONNEMENT ET DE DISTRIBUTION DES GRAPHES

Sommaire

1.1 Introduction

Depuis le problème des ponts de Königsberg [EULER, 1736], la théorie des graphes s'est particulièrement développée en raison du nombre élevé de problèmes qu'elle permet de résoudre. C'est un outil privilégié de modélisation mathématique et de résolution de problèmes dans un grand nombre de domaines allant de la science fondamentale aux applications technologiques concrètes. Par exemple, les graphes déterministes et aléatoires sont utilisés en chimie (modélisation de structure), en sciences sociales (pour représenter des relations entre groupes d'individus), en mathématiques combinatoires, en informatique (structures de données et algorithmique). La modélisation mathématique facilite la compréhension d'un problème, car elle détermine un seul vocabulaire formel pour différentes situations, et elle permet de trouver une méthode de résolution efficacement et optimale. La théorie des graphes à une très large gamme d'applications dans divers domaines, en particulier chez les mathématiciens et les ingénieurs [DEO, 2017].

En informatique, les graphes jouent un rôle important dans de nombreuses branches ; ils sont utilisés pour représenter les interactions statiques ou dynamiques dans des réseaux complexes (interactions entre amis dans un réseau social ou son évolution, l'activation des gènes dans les systèmes biologiques, des réactions chimiques dans les réseaux biochimiques). En général, un graphe peut être utilisé pour représenter toute situation impliquant des objets discrets et des relations entre eux ; les graphes sont ensuite analysés pour découvrir les propriétés de l'objet modélisé, ou transformés pour construire d'autres types de modèles.

Nous présentons, dans ce chapitre, quelques notions mathématiques et de l'optimisation combinatoire, nous introduisant quelques notions sur la théorie des graphes et les différentes approches de partitionnement et de distribution des graphes.

1.2 Notions mathématiques

Cette section rappelle quelques définitions mathématiques.

Définition 1 (Ensemble)**.** Un ensemble désigne une collection d'éléments, la notation utilisé pour décrire un ensemble est S = $\{s1, ..., sn\}$. Les ensembles utilisés dans ce document sont finis et leur taille est notée | S |. On note $\emptyset$ l'ensemble vide.

Définition 2 (Cardinal d'un ensemble)**.** Soit X un ensemble fini d'éléments. Le cardinal de l'ensemble X est égal au nombre éléments de X, et on le note card(X).

Définition 3 (Partition)**.** Soit un ensemble S quelconque. Un ensemble P de sous-ensembles de S est appelé une partition de S si :

— Aucun élément de P n'est vide ;

— L'union des éléments de P est égale à S ;

— Les éléments de P sont deux à deux disjoints.

Les éléments de P sont appelés les parties de la partition P. Le cardinal de la partition P est alors le nombre de parties de P.

Le partitionnement de graphe fait partie des problèmes d'optimisation combinatoire, qui est une branche des mathématiques discrètes. Ce dernier parfois appelées mathématiques finies, sont l'étude des structures mathématiques où la notion de continuité est absente. Les objets étudiés en mathématiques discrètes (tels que les entier relatifs, les

graphes simples et les énoncés en logique) sont des ensembles dénombrables [L, 1989]. Dans le cadre des mathématiques discrètes, un ensemble dénombrable (ensembles qui ont la même cardinalité que les sous-ensembles des nombres naturels, y compris les nombres rationnels mais pas les nombres réels) peut aussi être appelé ensemble discret.

1.3 Optimisation combinatoire

L'optimisation combinatoire, appelée aussi optimisation discrète, est une branche de l'optimisation en mathématiques appliquées et en informatique, également liée à la recherche opérationnelle, l'algorithmique et la théorie de la complexité, elle consiste à trouver un *meilleur* choix parmi un ensemble fini (souvent très grand) de possibilités. Elle recouvre les méthodes qui servent à déterminer l'optimum ou montrer la difficulté de résoudre une fonction sous des contraintes données, contrairement aux fonctions sans contrainte la solution optimale correspond au coût optimal (minimal, maximal) de la fonction.

La plupart des problèmes d'optimisation appartiennent à la classe des problèmes NP-difficile classe où il n'existe pas d'algorithme qui fournit la solution optimale en temps polynomial en fonction de la taille du problème et le nombre d'objectifs à optimiser. D'où la nécessité d'utiliser les méthodes approchées (Méta heuristique, Heuristique, etc.) pour obtenir l'ensemble des solutions admissibles aux problèmes. Dans ce qui suit nous présentons les concepts et vocabulaires liés au domaine.

Définition 4 (Fonction objectif)**.** Une fonction objectif est une fonction qui modélise le but à atteindre dans le problème d'optimisation sur l'ensemble des critères. Il s'agit de la fonction qui doit être optimisée. Elle est notée F(x) de manière générale F(x) est un vecteur : F(x) = [$f1(x), f2(x), ..., fk(x)$]. Elle est aussi appelée : critère d'optimisation, fonction coût, fonction d'adaptation, ou encore performance.

Définition 5 (Paramètres)**.** Un paramètre du problème d'optimisation, est une variable qui exprime une donnée quantitative ou qualitative sur une dimension du problème : coût, temps, taux d'erreurs, etc. Ces paramètres correspondent aux variables de la fonction objective. Ils sont ajustés pendant le processus d'optimisation, pour obtenir les solutions optimales. On les appelle aussi variables d'optimisation, variables de conception ou de projet.

Définition 6 (Vecteur de décision)**.** Un vecteur de décision est un vecteur correspondant à l'ensemble des variables du problème, il est noté : $\vec{x} = [x_1, x_1 2, x_3, \ldots, x_n]^{\mathrm{T}}$ avec : n le nombre de variables ou dimension du problème et x_k la variable sur la dimension K.

Définition 7 (Critère de décision)**.** C'est un critère sur lequel sont jugés les vecteurs de décision pour déterminer le meilleur vecteur. Un critère peut être une variable du problème ou une combinaison de variables.

Définition 8 (Contraintes)**.** Une contrainte du problème est une condition que doivent respecter les vecteurs de décision du problème. Une contrainte est notée : $g_i(\vec{x})$ avec $i = 1, \ldots, q$, q : le nombre des contraintes

Définition 9 (Solution admissible)**.** Une solution admissible est un ensemble de valeurs données aux variables qui satisfait toutes les contraintes.

Définition 10 (Espace de recherche)**.** L'espace de recherche représente l'ensemble des valeurs qui peuvent être prises par les variables.

Définition 11 (Solution optimale)**.** Une solution optimale est une solution admissible qui optimise la fonction objectif.

Définition 12 (Problème d'optimisation combinatoire)**.** Un problème d'optimisation combinatoire se définit à partir d'un triplet (E, p, f) tel que :

- E est un ensemble discret appelé espace des solutions (aussi appelé espace de recherche) ;
- p est un prédicat sur E, i.e. une fonction de E dans vrai, faux ;
- $f : S \rightarrow IR$ associe à tout élément $x \in E$ un coût $f(x)$. f est appelée fonction de coût ou fonction objectif.

p permet de créer un ensemble $E_a = \{x \in E$ tel que $P(x)$ est vrai $\}$. L'ensemble E_a est appelé l'ensemble des solutions admissibles du problème.
Il s'agit de trouver un élément $\tilde{x} \in E_a$ qui minimise f :

$$f(\tilde{x}) = \min_{x \in E_a} f(x)$$

Lorsque le problème d'optimisation combinatoire consistant à chercher un élément maximum au lieu d'un élément minimum on a :

$$\max_{x \in E_a} f(x) = -\min_{x \in E_a}(-f(x))$$

Les problèmes d'optimisation combinatoire sont en général très coûteux à résoudre de façon optimale. C'est en particulier le cas du partitionnement de graphe. Lorsque le problème n'est pas soumis à aucune contraintes, le problème d'optimisation combinatoire, vise à trouver une partition des sommets d'un graphe $G = (S, A)$ en k parties de tailles égales (on choisit k diviseur de $card(S)$), aura pour ensemble de solutions E l'ensemble des partitions de S dont le nombre de parties va de un au nombre d'éléments de S, et dont les parties sont de tailles quelconques. Par contre, l'ensemble des solutions admissibles du problème, E_a, doit tenir compte des contraintes de celui-ci.

Définition 13 (Optimum global, optimum local)**.** Soit un problème d'optimisation combinatoire (S, p, f) et S_a l'ensemble des solutions admissibles du problème induit par p. Soit $\tilde{x} \in S_a$.

- Si l'on peut prouver que $\forall x \in E_a$, $f(\tilde{x}) \leq f(x)$, alors on dira que $\tilde{x}$ est l'optimum (minimum) global du problème ;
- S'il existe un ensemble $V \subset E_a$, contenant $\tilde{x}$, et au moins deux éléments, tel que $\forall x \in V$, $f(\tilde{x}) \leq f(x)$, alors on dira que $\tilde{x}$ est un optimum (minimum) local du problème.

L'espace des solutions E dispose d'une « topologie ». Connaître les caractéristiques de celle-ci est très utile pour comprendre le but du fonctionnement des méta-heuristiques. Cette topologie résulte de la notion de proximité entre deux solutions, aussi appelées dans ce cas configurations. La distance entre deux configurations représente le nombre minimum de modifications élémentaires nécessaires pour passer de l'une à l'autre. De plus, puisqu'à chaque configuration x est associée une valeur $f(x)$, l'espace des solutions est caractérisé par une courbe à plusieurs dimensions appelée « paysage énergétique ». Dans ce paysage énergétique, les optimums locaux ou globaux forment des « puits énergétiques » autour d'eux. Avant de décrire une solution du problème comme étant un minimum local, on vérifie en général que l'ensemble V est suffisamment « grand » par rapport à la taille de E_a. La Figure 1.1 représente l'équivalent en continu du paysage énergétique d'une fonction de coût pour un espace des solutions à une dimension.

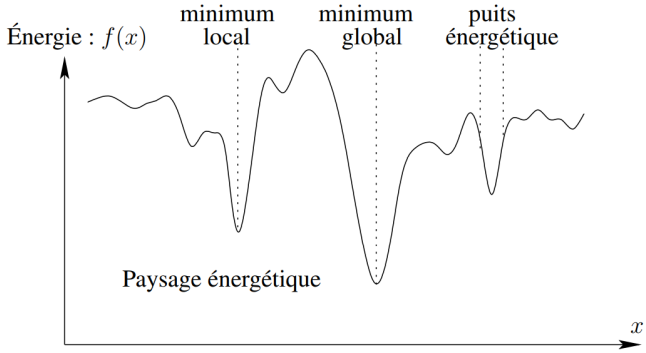


FIGURE 1.1 – Paysage énergétique dans le cadre continu d'une fonction de coût pour un espace des solutions à une dimension [BICHOT, 2007]

1.4 Graphes

Cette section rappelle quelques définitions de la théorie des graphes [QUEYROI, 2013].

Définition 14 (Graphe)**.** Un graphe est un couple G = (S, A), formé d'un ensemble S de sommets (ou nœuds ou points) et d'un ensemble A d'arêtes, d'arcs (ou lignes) qui sont associés à des sous-ensembles à deux éléments de S. Les sommets appartenant à une arête sont ses extrémités. Un sommet d'un graphe n'est pas nécessairement extrémité d'une arête : c'est alors un sommet isolé. La taille d'un graphe est, selon le cas, le nombre de ses sommets | S | ou le nombre de ses arêtes | A |. Selon la nature de l'ensemble A, le graphe peut être orienté ou non orienté.

Définition 15 (Graphe orienté, non orienté)**.** Soit un graphe G = (S, A). Si $\forall (x, y) \in$ A, $(y, x) \in$ A, alors le graphe est dit non orienté et les éléments de A sont appelés arêtes du graphe. Dans ce cas, on note indifféremment une arête : $a \in$ A, $(s, s') \in$ A avec s et s' dans S, ou encore (s', s). Dans le cas contraire, le graphe est dit orienté et les éléments de A sont appelés arcs du graphe.

Définition 16 (Adjacence)**.** Soient un graphe G = (S, A) et une arête $a = (s, s') \in$ A. On dit que les sommets s et s' sont les sommets adjacents (voisins) à l'arête a. De même, on dit que a est l'arête adjacente aux sommets s et s'.

Définition 17 (Degré d'un sommet)**.** Dans un graphe non orienté G = (S, A), le degré d'un sommet $s \in$ S est le nombre d'arêtes auxquelles ce sommet appartient :

$$deg(s) = card((s, s') \in \text{A}, s' \in \text{S})$$

Définition 18 (Graphe Complet)**.** Un graphe est complet si tous ses sommets sont adjacents entre eux. Deux sommets non adjacents peuvent malgré tout avoir une certaine connections. Dans ce cas, nous parlerons de chemin.

Définition 19 (Chemin)**.** Un chemin dans G = (S, A) de longueur n est une suite de sommets $a_1 a_2 ... a_n a_{n+1}$ reliés entre eux par des arêtes, c'est-à-dire $a_i a_{i+1} \in$ A, pour $i = 1, 2, ..., n$. Un cycle est un chemin fermé, C'est-à-dire que $a_1 = a_{n+1}$.

Définition 20 (Graphe connexe)**.** Soit un graphe G = (S, A). On dit que ce graphe est connexe si, quels que soient les sommets s et s' de S, il existe un chemin de s vers s'.

Définition 21 (Boucle et arête multiple)**.** Une arête est appelée une boucle si ses deux extrémités sont identiques. Si deux arêtes possèdent les mêmes extrémités, alors on dit que l'arête est multiple et que ces deux arêtes sont parallèles. Dans ce cas, la multiplicité d'une arête est le nombre total de ses arêtes parallèles, y compris elle-même.

Définition 22 (Graphe value ou pondéré)**.** Un graphe pondéré est un graphe étiqueté où chaque arêtes (arcs) est affectée d'un nombre réel positif, appelé poids de cette arête (arc).

Définition 23 (Sous-graphe)**.** Un sous-graphe $H = (S_H, A_H)$ d'un graphe $G = (S, A)$ est le graphe G auquel des sommets et/ou des arêtes ont été enlevés, c'est-à-dire

$$S_H \subseteq S \text{ et } A_H \subseteq A.$$

Définition 24 (Chaîne)**.** Une chaîne est une suite d'arcs partant d'un sommet s_1 et se terminant à un sommet s_n. Nous pouvons la définir comme suit :

$G = (S, A) | S = \{s_1, s_2, ..., s_n\} \wedge A = \{a_1, a_2, ..., a_{n-1}\}$ où $a_i = (s_i, s_i + 1)$ $i = 1..n-1$.

Définition 25 (Circuit)**.** Un circuit est une suite d'arcs partant et finissant au même sommet. Nous pouvons le définir comme suit :

$$G = (S, A) | S = \{s_1, s_2, ..., s_n\} \wedge A = \{(s_1, s_2), (s_2, s_3), ..., (s_{n-1}, s_n), (s_n, s_1)\}$$

Définition 26 (Appariement)**.** Soit un graphe $G = (S, A)$. L'appariement M du graphe G est un ensemble d'arêtes non-adjacentes deux à deux. On dit que l'appariement est maximum lorsqu'il contient le plus grand nombre possible d'arêtes. On dit que l'appariement M est maximal lorsque toute arête du graphe possède une intersection non vide avec au moins une arête de M.

Remarque Tout appariement maximal est aussi un appariement maximum. Soit un appariement maximal M, si une arête quelconque de A qui n'est pas dans M est ajoutée à M, alors M n'est plus un appariement de G. La réciproque n'est pas nécessairement exacte : soit un graphe linéaire de 4 sommets, l'appariement composé de l'arête formée des deux sommets centraux est maximal mais pas maximum. Un graphe peut posséder plusieurs appariements maximal (et a fortiori plusieurs appariements maximum).

1.5 Théorie de jeux

La théorie de jeux appelée aussi théorie de décision en interaction est une domaine qui inspire des mathématiques (probabilités, optimisation/contrôle, combinatoire, logique, calculabilité, complexité) et d'autre science tel que économie, cryptographie, physique quantique, cybernétique, biologie, sociologie, linguistique, philosophie. Les premiers fondements de ce domaine étaient décrits autour des années 1920 par Ernst Zermelo [**?**], Émile Borel [**?**] et John von Neumann [**?**]. Les idées de la théorie des jeux sont ensuite développées par Oskar Morgenstern et John von Neumann en 1944 [**?**].

La théorie de jeux se propose d'étudier des situations (appelées jeux) où des individus (les joueurs) prennent des décisions, chacun étant conscient que le résultat de son propre choix (ses gains) dépend de celui des autres. Ces décisions ayant pour but un gain maximum ou un gain stabilisé.

Dans ce qui suit nous présentons les concepts qui lui sont propres.

Définition 27 (Stratégie)**.** La stratégie d'un joueur est la fonction par laquelle il choisit son coup de jouer en fonction de l'état du jeu (ou de la fonction de l'état qui lui est présentée). Ainsi un jeu est constitué d'un ensemble de stratégies et de règles du jeu. Une stratégie est dite gagnante lorsque le joueur qui l'utilise gagne le jeu (supposé avoir une notion de « joueur gagnant ») quels que soient les coups choisis par l'autre joueur.

Définition 28 (Stratégie pure)**.** Une stratégie pure du joueur J_i est un plan d'action qui prescrit une action de ce joueur pour chaque fois qu'il est susceptible de jouer. L'ensemble des stratégies pures du joueur J_i est noté par S_i.

Définition 29 (Stratégie mixte)**.** Une stratégie mixte du joueur J_i est une distribution de probabilités p_i définie sur l'ensemble des stratégies pures du joueur J_i. L'ensemble des stratégies mixtes du joueur J_i est noté note $\sum_i$.

Définition 30 (Stratégie locale)**.** Une stratégie locale du joueur J_i est un ensemble d'information A et une distribution de probabilités sur l'ensemble des actions disponibles en cet ensemble d'information. L'ensemble des stratégies locales du joueur J_i pour l'ensemble d'information A est π_{iA}.

Définition 31 (Stratégie comportementale)**.** Une stratégie comportementale du joueur J_i est un vecteur de stratégies locales de ce joueur, contenant une stratégie locale par ensemble d'information de ce joueur. On note π_i l'ensemble des stratégies comportementales du joueur J_i.

Définition 32 (Jeux sous forme stratégique)**.** Un jeu sous forme stratégique est défini par :

- Un ensemble $N = \{1, ..., n\}$ de joueurs.
- Pour chaque joueur J_i un ensemble de stratégies $S_i = \{s_1, ..., s_n\}$.
- Pour chaque joueur J_i une fonction dévaluation $\mu_i : s_1 \times ... \times s_n \longrightarrow IR$, qui à chaque ensemble de stratégies associe les gains du joueur J_i.

Définition 33 (Profil stratégique)**.** Un profil stratégique $(s_1*, \ldots, s_n*)$ est une solution d'un jeu, on peut justifier que des joueurs rationnels, guidés par leur intérêt personnel, le joueraient.

Définition 34 (Stratégie dominante)**.** Un stratégie s_i* d'un joueur est une stratégie dominante lorsque pour tout profil de stratégies des autres joueurs, le gain du joueur est maximum lorsqu'il joue cette stratégie.

Définition 35 (Équilibre)**.** On dit qu'un jeu possède un équilibre en stratégies dominantes s'il admet un profil stratégique composé uniquement de stratégies dominantes des joueurs.

Définition 36 (Équilibre de Nash)**.** On dit qu'un jeu possède un équilibre de Nash s'il admet un profil stratégique $(s_1*, \ldots, s_n*)$, tel que chaque stratégie individuelle de ce profil est une meilleure réponse aux stratégies des autres joueurs.

1.6 Partitionnement de graphe

Le partitionnement de graphe est un problème NP-Complet [GAREY, 1976], utilisé pour résoudre un grand nombre de problèmes d'ingénierie : La conception de circuits intégrés électroniques [CONG, 2003], la répartition de charge pour les machines parallèles

[BARAT, 2016], la dynamique des fluides, le calcul matriciel, la segmentation d'images [GRADY, 2006] ou la classification d'objets [DHILLON, 2007].

Etant donné un graphe non-orienté G = (S, A) où S est l'ensemble des sommets et A est l'ensemble des arêtes. Les sommets et les arêtes peuvent être pondérés. Le problème du partitionnement d'un graphe G consiste à le diviser en k parties disjointes. Du point de vue mathématique, on peut partitionner les sommets ou bien les arêtes. Cependant, bien que certains problèmes cherchent à partitionner les arêtes d'un graphe [HOLYER, 1981], on entend le plus souvent par partition d'un graphe, le partitionnement des sommets de ce graphe.

Définition 37 (Partition des sommets d'un graphe)**.** [BICHOT, 2013]
Soient un graphe G = (S, A) et un ensemble de k sous-ensembles de S, noté $P_k = \{S_1, S_2, ..., S_n\}$. En d'autres termes, P_k est dite une partition de G si :

- Aucun sous-ensemble de A qui est élément de P_k n'est vide : $\forall i \in \{1, ..., k\}$, $S_i \neq \emptyset$
- Les sous-ensembles de S qui sont éléments de P_k sont disjoins deux à deux : $\forall (i, j) \in 1, ..., K, i \neq j, S_i \cap S_j = \emptyset$
- L'union de tous les éléments de P_k est S : $U_{i=1}^{k} S_k = S$

Les éléments S_i de P_k sont appelés les parties de la partition.
Le nombre k est appelé le cardinal de la partition, ou encore le nombre de parties de la partition. Dans le cas $k = 2$, on a le problème de bissection.

1.7 Approches de partitionnement

Le partitionnement d'un graphe consiste à trouver une partition des sommets respectant une ou plusieurs propriétés. Ces propriétés sont de nature très diverses. On peut par exemple chercher à minimiser la capacité des arêtes externes, c'est-à-dire le nombre d'arêtes ou la somme totale des poids des arêtes reliant deux groupes. Le problème devient difficile si on cherche à diviser le graphe en k groupes de tailles équilibrées car il n'existe alors pas d'algorithme polynomial pour y répondre. Ce problème a été démontré NP-complet [HYAFIL, 1973], [GAREY, 1976]

Au cours des dernières années, beaucoup d'efforts ont été consacrés au développement d'heuristiques rapides et efficaces pour ce problème. Ces heuristiques peuvent souvent gérer des graphes assez grands avec des milliers de nœuds et fournir de bonnes solutions. Pour un aperçu plus détaillé, nous nous référons à [BICHOT, 2013], [BULUÇ, 2016],[SCHLOEGEL, 2000].

Nous allons maintenant présenter les approches couramment utilisées pour résoudre le problème du partitionnement de graphes.

Approche exacte

Les solutions exactes peuvent être utilisées comme des facteurs importants pour valider les heuristiques. Cependant, peux d'effort ont été faits dans le développement d'algorithmes exacts [ARMBRUSTER, 2008], [BONAMI, 2012], [DELLING, 2012], [FELDMANN, 2015], [FERREIRA, 1998], [HAGER, 2013], [KAIBEL, 2011], [SELLMANN, 2003], [SØRENSEN, 2003]. La plupart de ces méthodes reposent sur la méthode de séparation et d'évaluation qui permet d'éviter l'exploration systématique de l'espace des solutions en éliminant les sous-arbres menant à des solutions moins bonnes qu'une solution donnée, et se diffèrent selon que l'exploration de l'arbre est réalisée en profondeur, par niveaux ou

par ordre croissant des valeurs des fils non encore traités [LAND, 2010]. Du fait de la nature NP-difficile du problème, il est clair que généralement seuls des graphes relativement petits peuvent être résolus par une approche exacte.

Approche spectrale

L'approche spectrale du partitionnement de graphes repose sur le théorème spectral de l'algèbre linéaire, elle consiste à rechercher les valeurs propres de la matrice de Laplace associée au graphe G, ensuite l'ordonnancement de ces valeurs qui est lié à la connectivité des sommets du graphe. Les techniques spectrales ont d'abord été utilisées par Donath et Hoffman [DONATH, 1972] et Fiedler [FIEDLER, 1973], [FIEDLER, 1975], et ont été améliorées par la suite par plusieurs chercheurs [BOPPANA, 1987], [HENDRICKSON, 1995a], [KABELÍKOVÁ, 2006], [SIMON, 1991]. Il a été montré que cette méthode permet d'obtenir un extremum global avec certains graphes [POTHEN, 1990] ; cependant, il a aussi été mis en évidence que cette méthode est très coûteuse en terme de calculs et devient inefficaces lorsque la taille du graphe devient importante [BARNARD, 1994].

L'approche combinatoire

[BENSETIRA, 2017] Contrairement à l'approche spectrale, les travaux cités dans cette approche opèrent directement sur la structure du graphe. L'idée générale consiste à effectuer un parcours de proche en proche d'une partie de l'ensemble P(G) afin d'y trouver le meilleur candidat qui résolve le problème. En plus, un algorithme de ce type nécessite les deux données suivantes :

— la définition d'un voisinage dans P(G) ;

— l'historique des optimisations précédentes.

Le voisinage permet de définir comment l'algorithme va progresser en perturbant la solution courante S : par exemple, on peut définir le voisinage de S comme correspondant à un seul déplacement de sommet par l'ensemble des éléments de P(G) qui peuvent être obtenus en changeant dans la partie un seul sommet de S.

Dans cette approche, on distingue entre les méthodes basées sur les algorithmes itératifs et les méthodes basées sur les méta-heuristiques.

Les algorithmes itératifs

Les algorithmes itératifs d'optimisation fonctionnent en partant d'une partition $\prod_0 \in$ P(G) de G, valide et bien équilibrée, et se déplacent dans l'espace des solutions en sélectionnant le voisin le plus proche qui permet de réduire la coupe de la partition. L'algorithme s'arrête lorsqu'aucun des voisins n'est satisfaisant. Ces algorithmes convergent donc vers l'optimum local accessible depuis la partition initiale $\prod_0$.

Bien que la convergence ne soit que locale et dépendante du point de départ dans P(G), ces algorithmes sont très populaires. Les résultats produits pouvant être améliorés en effectuant plusieurs exécutions à partir de partitions initiales différentes ou en utilisant le schéma multi-niveaux qui sera présenté plus loin. Parmi les algorithmes de cette catégorie, nous pouvons citer [FIDUCCIA, 1988], [HAGEN, 1997], [KERNIGHAN, 1970], [KRISHNAMURTHY, 1984].

Le principal défaut de ces approches itératives concerne leur vision strictement locale du problème, ce qui implique que le résultat ne peut être au mieux qu'un optimum local,

fortement dépendant de la solution initiale. D'autres approches d'algorithmes d'optimisation ayant une vision plus globale peuvent être utilisées.

Les approches basées sur les méta-heuristiques

Les algorithmes génétiques, ont fait l'objet de plusieurs approches pour le problème du partitionnement de graphes [BUI, 1996], [CHEVALIER, 2006], [CHOCKALINGAM, 1992], [SOPER, 2004], [TALBI, 1991]. Le principe est d'explorer l'espace des solutions à l'aide d'une population d'individus qui vont échanger entre eux certaines de leurs caractéristiques lors d'une étape de reproduction, ceci dans le but d'obtenir de nouveaux individus combinant les qualités de leurs parents tout en essayant d'en diminuer les défauts. Pour un aperçu général des algorithmes génétiques abordant le problème du partitionnement de graphe, nous renvoyons le lecteur à l'article de [KIM, 2011].

Les principales difficultés d'application de ces algorithmes tiennent d'une part à la multitude de paramètres qu'il faut régler en fonction de la nature du problème, d'autre part à leurs coûts en mémoire et en temps : La taille de la population est liée à celle du problème et le temps est dépendant du nombre de générations qui est lié à la taille de la population.

Les méthodes de colonies de fourmis consistent à imiter le comportement des fourmis qui exploitent une méthode de résolution collective. En effet, les fourmis utilisent des phéromones pour marquer les différents chemins empruntés et la concentration de ces marqueurs chimiques est d'autant plus importante que la fréquence d'utilisation est importante. Pour le k-partitionnement, l'idée est donc d'utiliser k colonies de fourmis dont le but est d'accumuler la nourriture qui est distribuée sur les sommets du graphe, l'exploration est guidée par la répartition des colonies, de la nourriture et par les décisions des fourmis [KOROŠEC, 2004], [LANGHAM, 1999], [JOVANOVIC, 2016].

D'autres méta-heuristiques, comme la recherche tabou [BATTITI, 1999], le recuit simulé [BICHOT, 2004], [JOHNSON, 1989], [WILLIAMS, 1991], les algorithmes de gradients aléatoires adaptatifs (GRASP) [AREIBI, 2004], les méthodes de diffusion stochastique [TAO, 1993], [WAN, 2005], ont été appliquées avec plus ou moins de succès au partitionnement de graphe.

On peut aussi citer les travaux basés sur la méthode de percolation [MEYERHENKE, 2006], [PELLEGRINI, 2007] qui s'inspire du principe de l'écoulement d'un fluide à travers un solide poreux, et la méthode de fusion-fission [**?**] qui s'inspire de la physique nucléaire et plus particulièrement des fusions et fissions d'atomes. L'efficacité des méthodes combinatoires est étroitement liée à la taille de l'espace de recherche, les heuristiques calculent un partitionnement acceptable en un temps polynomial.

L'approche multi-niveaux

La méthode multi-niveaux, introduite par [BARNARD, 1994], [HENDRICKSON, 1995b], [VAN DRIESSCHE, 1994], permet de réduire la taille du graphe ainsi que celle de l'espace de recherche tout en procurant l'accès à une vision globale pour les algorithmes combinatoires. L'idée principale est de travailler sur un graphe réduit ayant les mêmes propriétés topologiques que le graphe initial (regrouper les sommets ensemble pour traiter de groupes de sommets plutôt que de sommets indépendants). Pour ce faire, le schéma multi-niveaux comporte trois étapes (Figure 1.2) :

1. L'étape de contraction, durant laquelle on applique récursivement une fonction de contraction afin de diminuer la taille du graphe ;

2. L'étape de partitionnement initial, où l'on applique une heuristique sur le plus petit graphe ;
3. L'étape d'affinage, durant laquelle la partition initiale est projetée et raffinée sur les graphes de plus en plus gros jusqu'au graphe initial.

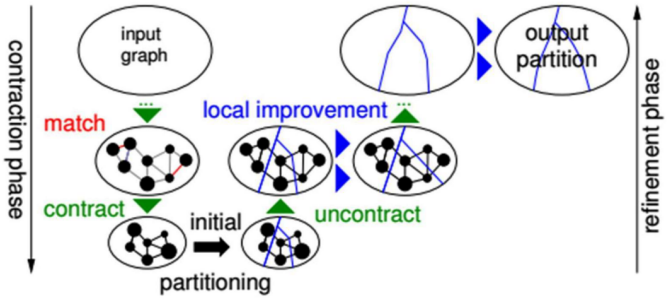


FIGURE 1.2 – L'approche multi-niveau du partitionnement de graphe, [SANDERS, 2011]

Dés lors, plusieurs améliorations ont été proposées [AURORA, 2007], [CHEVALIER, 2009], [KALAYCI, 2018], [KARYPIS, 2003], [KARYPIS, 1998], [MONIEN, 2000], [PELLEGRINI, 1995], [POPE, 2000], [PREDARI, 2017], [SAFRO, 2015], [SANDERS, 2011], [TALU, 2017].

La parallélisation du partitionnement de graphe

Des tentatives de parallélisation des algorithmes de partitionnement de graphes ont vu le jour. On peut citer ParMeTiS [KARYPIS, 2011], ParJostle [WALSHAW], PT-Scotch [CHEVALIER, 2008], et Parkway [TRIFUNOVIC, 2015] pour le partitionnement d'hyper-graphes. Tous ces outils utilisent un schéma multi-niveaux parallèle.

Des approches qui combinent les différentes méthodes présentées ci dessus ont été proposées, [CHAN, 2016], [LASALLE, 2013], [LASALLE, 2015], [LENG, 2007], [RAHIMIAN, 2013], [SANDERS, 2012], [TASHKOVA, 2011].

1.8 Distribution de graphes

Le contexte général de notre projet de fin d'étude est la vérification formelle des systèmes. Les systèmes à vérifier sont représentés par des modèles de spécification formels, qui génèrent sous une sémantique formelle des espaces d'états. Un espace d'états peut être considéré comme un graphe. Dans ce contexte particulier, on parle de distribution de graphes (espaces d'états), et non pas de partitionnement de graphe. Les différences principales entre eux résident dans le fait que :

— Le partitionnement des graphes opèrent sur des graphes simples non orientés et pondérés. Alors que dans notre contexte, les graphes sont orientés, non pondérés, et contiennent dans la plupart des cas des cycles.

— En générale, les applications qui utilisent le partitionnement des graphes supposent l'existence préalable du graphe. Tandis que les espaces des états sont construits au moment de leurs distribution.

— La distribution d'un graphe n'interdit pas la duplication de certains de ses états (sommets) sur plusieurs parties distribuées du graphes.

L'espace d'états distribué

Dans cette section, nous définissons la structure d'un espace d'états distribué qui se compose de sous-graphes localisés dans les différentes machines disponibles sur le réseau.

Un espace d'état est une structure relationnelle qui représente le comportement d'un système (programme, protocole, réseau social, ...). Il représente tous les états possibles du système et les transitions entre eux. Un espace d'états est un graphe orienté G = (S, A) avec un ensemble de sommets S, un ensemble d'arcs A.

Définition 38. Soit M = $\{M_K\}_{K=1..N}$ N machines. Un espace d'états distribué, noté DiG, est un graphe avec une fonction de distribution f^k. DiG = (G; f^k)$_{k=1..N}$, tel que : G = (S, A) : est un graphe dirigé. f^k : G ⇒ G_k : est une application de G dans G k , tel que
$G_k = (S_k, A_k)$: $\{G_K\}_{1<k<N}$ est un ensemble de sous-ensembles appelés fragments G_k, tel que :
$U_{k=1}^{N} S_k = S \; et \; U_{k=1}^{N} A_k = A$

Définition 39. [BENSETIRA, 2017]
Un fragment G_k est définit par $G_k = (S_k, A_k)$ de tel sorte que :

- $S_k \subseteq S$ est un fragment des états de S dans la machine M_k tel que :
  - Aucun élément de S_k n'est vide : $\forall\, k \in 1, ..., N,\ S_k \neq \emptyset$
  - L'union de tous les éléments de S_k est égale à S : $U_{k=1}^{N} S_k = S$
- $A_k = A_K^L \cap A_R^K$ tel que $A_K^L \cap A_R^K = \emptyset$: l'ensemble des transitions internes et externes avec :
  - $A_K^L \subseteq S_K \times S_K$ est l'ensemble des transitions entre les états qui appartient à la même machine M_k (transitions locales ou internes)
  - $A_K^R \subseteq S_K \times (S/S_K) \cup (S/S_K) \times S_K = (s, s')$, tel que soit $(s \in S_K ets' \neq S_K) ou (s \neq S_K ets' \in S_K)$: est l'ensemble des transitions dont les origines se trouvent dans des machines locales et les états cibles (destination) sont dans des machines distantes (transitions externes) et inversement.

Le nombre k est appelé la cardinalité du fragment.

La Figure 1.3(a) représente un espace d'états, et une distribution possible de cet espace sur un réseau de trois machines (Figure 1.3(b)). Les transitions internes sont marquées avec des lignes continues, et les transitions externes sont marquées avec des lignes en pointillés.

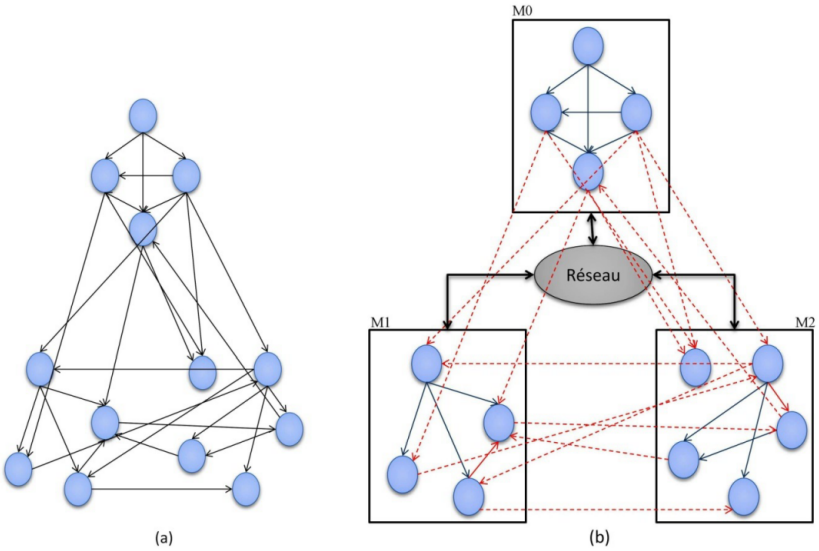


FIGURE 1.3 – Un espace d'état et sa distribution sur un cluster de 3 machines, [BENSETIRA, 2017]

Approche de distribution des espaces d'états

Distribution basée sur des fonctions La plupart des solutions proposées pour la distribution des espaces d'états reposent sur l'utilisation d'une fonction de distribution qu'elle soit statique [GARAVEL, 2013]ou dynamique [ALLMAIER, 1997], basée sur la structure de formalisme de spécification utilisé [CIARDO, 1998], [BLOM, 2005] ou guidée par des heuristiques [RODRIGUES, 2006].

Distribution basée sur la coloration Cette approche consiste à introduire le concept de coloration et la relation de dominance dans les graphes pour trouver la bonne distribution des graphes L'approche proposée dans [GUIDOUM, 2013] est basée sur le concept de la coloration forte stricte des graphes [BOUZENADA, 2012]. L'approche proposée est divisée en deux parties : un processus d'initialisation et un processus d'optimisation. Dans la première étape, les auteurs utilisent l'algorithme de coloration [BOUZENADA, 2012] qui assure la propriété de dominance. Les sorties de ce dernier (le nombre de couleurs dominées, ensemble de sommets colorés) sont exploitées pour répartir initialement le graphe et obtenir des parties disjointes. Ensuite le processus d'optimisation est utilisé pour trouver la bonne distribution.

Distribution basé sur les métaheuristiques Récemment une nouvelle alternative de distribution des espaces d'états a été investiguée. Les approches proposées dans cette classe se basent sur l'utilisation des méthodes d'optimisation combinatoire, qui ont montré leur efficacité de résolution des problèmes dans différents domaines d'application.

Saidouni et al. [SAIDOUNI, 2012] proposent un nouvel algorithme de distribution inspiré de l'algorithme d'optimisation par essaims de particules. Les auteurs ont pu montrer comment les composants de la métaheuristique PSO comme le voisinage, les particules, et leurs vitesses et direction peuvent être adaptés pour la distribution des graphes sur une plateforme de N machines. L'approche fournit un bon équilibrage de charge et une réduction des connexions externes dans le réseau. Cependant, cette approche souffre du problème des états redondants.

Une autre approche a été développée dans [TABIB, 2016], [TABIB, 2017], où les auteurs ont proposé un algorithme génétique basé sur la loi gravitationnelle de Newton pour la distribution de graphes. La particularité de cette approche est que les espaces d'états sont codés par les diagrammes de décision binaires, et l'utilisation d'un modèle génétique en ilot qui repose sur l'exécution parallèle de plusieurs machines qui contiennent des populations de même taille avec la migration de leur m meilleurs individus chaque n cycles.

Saidouni et Bensetira [BENSETIRA, 2017], proposent un nouvel algorithme de distribution de l'espace d'états basée sur le comportement des systèmes. La particularité de cette approche est que les états sont analysés, ensuite, selon les informations pertinentes sur les états et leurs connexions (transitions internes et externes), ils sont redistribués selon une certaines heuristiques afin d'optimiser les performances du système. Cela se fait en définissant une bonne localité d'un état comme celle qui optimise à la fois l'équilibre de la charge de travail et la quantité de communication entre les machines engendrés par l'exécution de l'application opérant sur l'espace d'états distribués.

1.9 Conclusion

Dans ce chapitre, nous avons introduit quelques notions sur les graphes, ensuite nous avons donné un bref aperçu du travail qui a été fait sur le partitionnement et la distribution des graphes. Il nous est alors possible de développer de nouveaux algorithmes de distribution de graphe en exploitant le model cheching et les méthodes d'optimisation combinatoire.

Chapitre 2

MODEL CHECKING

Sommaire

2.1 Introduction

De nos jours, la vie des êtres humains dépend largement des systèmes informatiques qui remplissent des fonctions de plus en plus critiques. Ceci à augmenter la difficulté d'assurer le bon fonctionnement de ces systèmes afin d'éviter des conséquences qui peuvent s'avérer fatales, coûteuses et dramatiques [**?**], [**?**]. Dès lors, il faut fournir des techniques de vérification et de validation efficaces et performantes qui garantissent la sûreté de fonctionnement de ces systèmes critiques et qui prennent en charge leurs complexités croissantes [**?**].

Les méthodes formelles apportent une solution à ce problème, car elles permettent de définir des modèles mathématiques décrivant rigoureusement le comportement des systèmes. Elles les décrivent principalement à l'aide de langages formels adaptés et définis avec précision au niveau de la syntaxe et de la sémantique. Une fois décrit, elles peuvent prouver la correction du modèle en utilisant des méthodes de vérification formelle. La vérification consiste alors à s'assurer que l'ensemble des fonctionnements du système développé satisfait tous les besoins [**?**], [**?**].

Le model checking est une forme de vérification formelle des systèmes [**?**], [**?**]. Son principe est de vérifier que le modèle représentant le système, respecte bien les propriétés que l'on attend de lui, d'où le terme model checking. Les systèmes sont spécifiés par des formalismes tels que les automates temporisés [**?**], et les propriétés sont exprimées par des assertions dans une logique, par exemple la logique temporelle [**?**]. Bien que cette technique soit restreinte à des systèmes ayant un nombre fini d'états, elle permet une détection rapide et économique des erreurs dans des systèmes complexes. Parmi les outils les plus connus, on peut citer UPPAAL, DiVine, CADP, Murphi, Roméo, SPIN [**?**], [**?**], [GARAVEL, 2013], [**?**], [**?**], [**?**].

Les réseaux de Petri sont des outils à la fois graphiques et mathématiques permettant de modéliser le comportement dynamique des systèmes à évènements discrets. Leur représentation graphique permet de visualiser d'une manière naturelle le parallélisme, la synchronisation, le partage de ressources, les choix (conflits), etc. Leur représentation mathématique permet d'analyser le modèle pour étudier ses propriétés et de les comparer avec le comportement du système réel. L'une des approches de vérification d'un réseau Petri consiste à générer son graphe de marquage dont les sommets représentent les états dans lesquels le système peut se trouver, et dont les arcs représentent les transitions faisant passer le système d'un état à un autre. Après la génération, le graphe de marquage peut être vu comme un système de transitions étiquetées [**?**]. Le système de transitions étiquetées, ainsi généré, est utilisé pour la vérification des propriétés du système spécifié par le réseau (model checking, bissimilarité, test de conformité, etc. [**?**], [**?**], [**?**]). Cependant, le modèle des systèmes de transitions étiquetées fait abstraction quant à l'exécution parallèle des transitions.

Nous nous intéressons dans ce chapitre, au réseaux de Petri et au model checking distribuée. Nous présentons dans un premier lieu les réseaux de Petri, puis les systèmes de transitions étiquetées et aussi la structure de Kripke. Enfin, on termine avec le model checking distribué.

2.2 Réseaux de Petri

Les Réseaux de Petri (abréviation RdP) ont été introduits par le mathématicien Allemand Carl Adam Petri dans sa thèse [PETRI, 1962], d'où leur appellation Réseaux de Petri (ou Petri Nets en anglais). Ces derniers méritent bien leur appellation car la thèse de C.A

Petri a présenté un certain nombre d'idées fondamentales du modèle. Mais la théorie de RdP en sa totalité, telle que nous la connaissons actuellement est le résultat de fusionnement et de contribution directe ou indirecte des travaux de plusieurs chercheurs de différentes universités et de différents laboratoires [VALETTE, 9novembre 2007].

Les réseaux de Petri sont des outils à la fois graphiques et mathématiques permettant de modéliser le comportement dynamique des systèmes à évènements discrets et d'évolutions simultanées. Leur représentation graphique permet de visualiser d'une manière naturelle le parallélisme, la synchronisation, le partage de ressources, les choix (conflits), etc. Leur représentation mathématique permet d'analyser le modèle pour étudier ses propriétés et de les comparer avec le comportement du système réel.

Ce section présente les notions fondamentales des réseaux de Petri.

Définition 40 (Informelle)**.** Un réseau de Petri est un graphe biparti dont les sommets sont répartis selon deux types, les places et les transitions. Ce graphe est constitué de telle sorte que les arcs du graphe ne peuvent relier que des places aux transitions ou des transitions aux places. Les places sont représentées par des cercles, alors que les transitions sont représentées par des traits ou des rectangles (Figure 2.1). Les places servent à représenter les états du système modélisé, tandis que les transitions représentent les changements d'état ou les événements [**?**].

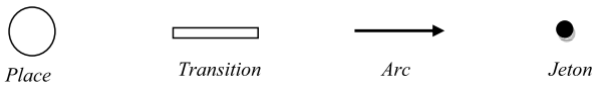


FIGURE 2.1 – Représentation graphique des éléments de RdP

Chaque place peut contenir un nombre entier de jetons. Les jetons modélisent souvent l'état d'une ressource (nombre d'instances, occupée ou non, ...). Un jeton est représenté par un petit cercle noir. Pour des commodités de présentation on met à l'intérieur d'une place le nombre de jetons présents (Figure 2.2).

FIGURE 2.2 – Marquage d'une place

À chaque arc est associé un nombre entier strictement positif appelé poids de l'arc. Lorsque le poids n'est pas signalé, il est égal à "1" par défaut. Le RdP dont tous ses arcs sont de poids "1" est appelé RdP ordinaire (Figure 2.3). Dans le cas où les arcs peuvent avoir des poids supérieurs à "1", il s'agit de RdP généralité (Figure 2.4).

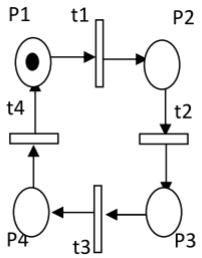


FIGURE 2.3 – RdP ordinaire

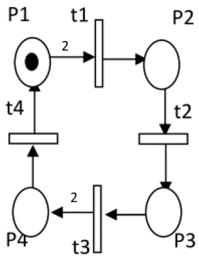


FIGURE 2.4 – RdP généralité

Définition 41 (Formelle)**.** [T, 1989]
Formellement un RDP est un quintuplet, R = (P, T, Pre, Post, M_0) tel que :

- P : L'ensemble des places ;
- T : L'ensemble des transitions ;
- Pre : (P × T) → IN, est l'application d'incidence avant, correspondant aux arcs directs reliant les places aux transitions ;
- Post : (T × P) → IN,est l'application d'incidence arrière, correspondant aux arcs directs reliant les transitions aux places ;
- La matrice incidence du réseau est C = Post − Pre ;
- M_0 : Le marquage initial (état initial).

Notation

- °t : L'ensemble des places d'entrée de la transition t ;
- t° : L'ensemble des places de sortie de la transition t ;
- °p : L'ensemble des transitions d'entrée de la place p ;
- P° : L'ensemble de transitions de sortie de la place p ;

Le RdP de la Figure 2.5 décrit le cycle des quatre saisons. Les places représentent les saisons, p_1 ; le printemps, p_2 ; l'été, p_3 : l'automne et p_4 : l'hiver. Les transitions représentent les changements de saisons, t_1 ; le début d'été, t_2 ; le début d'automne, t_3 : le début d'hiver et t_4 : le début du printemps. Le jeton dans la place p_1 indique que la saison à l'instant initial est le printemps. Ce scénario est modélisé comme suit :
P = {$p1, p2, p3, p4$},
T = {$t1, t2, t3, t4$},
$Pre(p_1, t_1) = 1,\ Post(t_1, p_2) = 1,\ Pre(p_2, t_2) = 1,\ Post(t_2, p_3) = 1,\ Pre(p_3, t_3) = 1,$
$Post(t_3, p_4) = 1,\ Pre(p_4, t_4) = 1,\ Post(t_4, p_1) = 1$,
$^\circ t_1 = \{p_1\},\ t_1^\circ = \{p_2\},\ ^\circ t_2 = \{p_2\},\ t_2^\circ = \{p_3\},\ ^\circ t_3 = \{p_3\},\ t_3^\circ = \{p_4\},\ ^\circ t_4 = \{p_4\},\ t_4^\circ = \{p_1\},$
$^\circ p_1 = \{t_4\},\ p_1^\circ = \{t_1\},\ ^\circ p_2 = \{t_1\},\ p_2^\circ = \{t_2\},\ ^\circ p_3 = \{t_2\},\ p_3^\circ = \{t_3\},\ ^\circ p_4 = \{t_3\},\ p_4^\circ = \{t_4\}.$

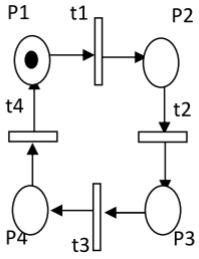


FIGURE 2.5 – Réseaux de Petri de quatre saisons

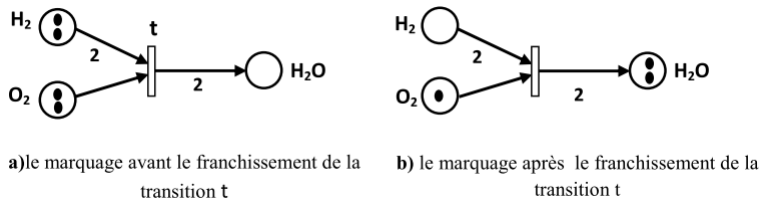
Définition 42 (Sensibilisation d'une transition)**.** Une transition t est dite sensibilisée (validée, franchissable ou tirable) si chacune des places d'entrée p contient un nombre de jetons supérieur ou égal au poids de l'arc reliant p à t [**?**].

$$\forall p \in \mathrm{P}, \mathrm{M}(p) \geq \mathrm{P}re(p,t)$$

Définition 43 (Franchissement d'une transition)**.** Le franchissement d'une transition t sensibilisée retire de chacune de ses places d'entrée p un nombre de jetons égal au poids de l'arc reliant p à t ($\mathrm{P}re(p,t)$) et dépose sur chacune de ses places de sortie p un nombre de jetons égal au poids de l'arc reliant t à p ($\mathrm{P}ost(p,t)$) [**?**]. Le franchissement d'une transition dans un marquage M donne un nouveau marquage M′ défini par :

$$\forall p \in \mathrm{P} : \mathrm{M}'(p) = \mathrm{M}(p) + \mathrm{C}(p,t)$$

La Figure 2.6 illustre la réaction chimique ($2H_2 + O_2 \rightarrow 2H_2O$) et le changement de marquage après le franchissement de la transition t. Avant le franchissement $M_0 = [2\ 2\ 0]$, après le franchissement $M_1 = [0\ 1\ 2]$. Les places sont ordonnées dans ce vecteur comme suit : H_2, O_2, H_2O, le marquage de la place H_2 est noté par $M(H_2)$. $M_0(H_2)$ est le marquage initial de la place H_2. Dans cet exemple $M_0(H_2) = 2$.



a)le marquage avant le franchissement de la transition t

b) le marquage après le franchissement de la transition t

FIGURE 2.6 – La composition d'eau ($2H_2 + O_2 \rightarrow 2H_2O$) sous forme d'un RdP

Remarque Lorsqu'une transition est validée, cela n'implique pas qu'elle sera immédiatement franchie; cela ne représente qu'une possibilité de franchissement, dans un RdP, même si plusieurs transitions sont validées par un même marquage une et seulement une transition peut être franchie.

Définition 44 (Graphe de Marquage)**.** Le graphe des marquages du réseau $G(R, M_0)$ est définit par un graphe orienté dont les sommets sont étiquetées par les marquages des états accessible $Acc(R, M_0)$ et dont les arcs sont étiquetés par des transition de $L(R, M_0)$. Un marquage M_i est accessible (atteignable) à partir de M_0 , s'il existe une séquence s de franchissement des transitions qui permet d'atteindre M_i à partir de M_0. On note : $M_0[s > M_i$

La construction du graphe de marquage G est faite comme suit [**?**] :

— Pour chaque marquage M obtenu à partir de M_0, trouver toute les transitions franchissable t_i ;

— Pour chaque transition t_i, trouver son marquage M′ ;

— Construire le nouveau nœud s'il est différent de celui déjà obtenu, puis ajouter l'arc correspondant au marquage actuel vers le prochain ;

— Continuer l'exploration tant que des marquages et des transitions n'ont pas été encore considérés.

La Figure 2.7 illustre le graphe de marquage obtenue à partir du RdP de la Figure Figure 2.6.

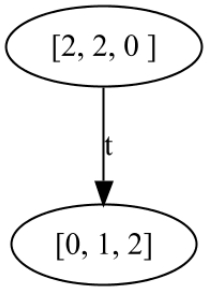


FIGURE 2.7 – Graphe de marquage de ($2H_2 + O_2 \rightarrow 2H_2O$)

Définition 45 (Système de transitions étiquetées)**.** Un système de transitions étiquetées appelé aussi STE est un graphe orienté dont les sommets et les arcs de ce graphe orienté sont appelés respectivement états et transitions. Ces états et transitions sont associés avec une chaîne de caractères de manière à pouvoir les distinguer entre eux. Ces chaînes de caractères s'appellent *noms* lorsqu'elles sont associées aux états, et étiquettes lorsqu'elles sont associées aux transitions .

Formellement un système de transitions étiquetées est un quadruplet $\text{STE} = (\text{S}, \text{A}ct, \delta, s_0)$ [SAIDOUNI, 2012] :

- **S** est un ensemble (dénombrable) d'états ;
- **Act** est un ensemble (dénombrable) d'actions dites observables ;
- $\delta \subseteq \text{S} \times (\text{A}ct \cup \{i\})$: est l'ensemble des transitions, $i \notin \text{A}ct$ est appelée action invisible (interne ou non-observable). Un élément $(x, a, y) \in \delta$ sera aussi noté par $x \xrightarrow{a} y$.
- $s_0 \in \text{S}$ est l'état initial de STE.

La Figure 2.8 illustre le STE du graphe de marquage de la Figure 2.7.

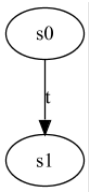


FIGURE 2.8 – STE de ($2H_2 + O_2 \rightarrow 2H_2O$)

Définition 46 (Structure de Kripke)**.** Une Structure de Kripke un graphe orienté dont les nœuds représentent les états accessibles du système et dont les arcs représentent les transitions entre les états. Une fonction d'étiquetage fait correspondre à chaque état un ensemble de propositions logiques vraies dans cet état [KRIPKE, 1963].

Formellement une structure de Kripke est un 4-uplet $\text{M} = (\text{S}, \text{I}, \text{R}, \text{L})$ [EDMUND M. CLARKE, 1999] :

- **S** est un ensemble fini d'états ;
- $\mathbf{I} \subseteq \mathbf{S}$ est un ensemble d'états initiaux ;
- $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{S}$ est une relation de transition qui vérifie : pour tout $s \in \text{S}$, il existe $s' \in \text{S}$ tel que $(s, s') \in \text{R}$;

  Soit AP un ensemble de propositions atomiques, c'est-à-dire des expressions booléennes portant sur des variables, des constantes et des prédicats. On note 2^{AP} l'ensemble des parties de AP.
- $\mathbf{L : S} \rightarrow 2^{\text{AP}}$ est une fonction d'étiquetage(ou d'interprétation) définit pour chaque état $s \in \text{S}$ l'ensemble $\text{L}(s)$ de toutes les propositions atomiques qui sont valides dans cet état.

Remarque La condition associée à la relation de transition R spécifie que chaque état doit avoir un successeur dans R, ce qui implique que l'on peut toujours construire un chemin infini dans la structure de Kripke. Cette propriété est importante lorsque l'on traite des systèmes réactifs[SCHNEIDER, 2004]. Pour modéliser un interblocage dans une structure de Kripke, il suffit de faire boucler l'état d'interblocage sur lui-même.

Un chemin dans la structure M est une suite $c = s_1, s_2, s_3, \ldots$ d'états tels que $(s_i, s_{i+1}) \in R$ pour tout i. L'étiquette du chemin est la suite d'ensembles $w = L(s_1), L(s_2), L(s_3), \ldots, \ldots$ qui peut être vu comme un mot infini sur l'alphabet 2^{AP}.

La Figure 2.9 représente une structure de Kripke dont l'ensemble de propositions atomiques est $AP = \{p, q\}$. Ici p et q sont des propriétés booléennes quelconques. L'état s_1 contient les deux propositions, les états s_2 et s_3 respectivement q et p. L'automate admet le chemin $c = s_1, s_2, s_1, s_2, s_3, s_3, s_3, \ldots$, et le mot $w = \{p, q\}, q, \{p, q\}, q, p, p, p, \ldots$ est la suite des étiquettes associées.

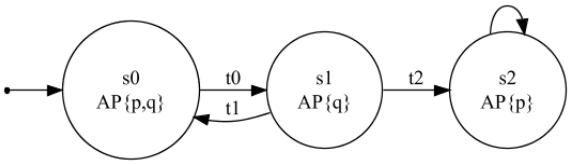


FIGURE 2.9 – Une structure de Kripke à trois états, avec deux propositions

La distribution de l'espace d'états entrain la fragmentation de la structure de Kripke. Les structures partielles de Kripke modélisent des espaces d'états incomplets à parties inconnues (états border). L'évaluation des formules logiques temporelles sur les structures partielles de Kripke repose donc sur des interprétations à trois valeurs ; la valeur de vérité supplémentaire $\perp$ signifie « inconnu si la propriété est vraie ou fausse ».
Formellement une structure partiale de Kripke est un 4-uplet
$F_M(T) = (S_T, I_T, R_T, L_T)$ [ABIDINE, 2011] :

— $T \subseteq S$;

— S_T est une sous ensemble d'états fini de l'espace d'états S ;

— $R_T \subseteq S_T \times S_T$ est une relation de transition qui vérifie : pour tout $s \in T$, il existe $s' \in S_T$ tel que $(s, s') \in R$;

— I_T est un ensemble fini d'états qui vérifie : pour tout $s \in S_T$ tel que $s \in I$} ;

— $L_T : S_T \times P \rightarrow \{false, \perp, true\}$.

La Figure 2.10 représente des structures partielles de Kripke repartie sur trois machines, dont l'ensemble de propositions atomiques est $AP = \{a, b, c\}$. La répartition des états est fait comme suit :

— Sur la machine M_1 la structure partiale renferme deux états à parties complets s_0 et s_1, et deux états à parties incomplets s_2 et s_4 ;

— Sur la machine M_2 la structure partiale renferme un état à partie complet s_4, et un état à partie incomplet s_5 ;

— Sur la machine M_3 la structure partiale renferme deux états à parties complets s_2 et s_5, et deux états à parties incomplets s_1 et s_4

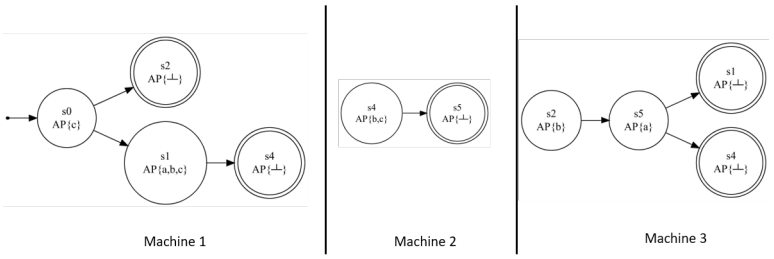


FIGURE 2.10 – Une structure de Kripke à trois états, avec deux propositions

2.3 Logique temporelle arborescente (CTL)

Logique temporelle définie des opérateurs permettant de « lier » des variables propositionnelles p aux instants t. Par exemple, une assertion liée au comportement d'un programme telle que « après exécution d'une instruction i, le système se bloque », les actions de cette assertion s'exécutent suivant un axe de temps : à l'instant t, exécution de l'instruction i, et à $t+1$ blocage du système. La logique du temps arborescent permet de modéliser les expressions du passé et du futur [J. W. BAKKER, 1983], [E. M. CLARKE, 1981], [E. A. EMERSON, 1983].

La logique du temps arborescent (CTL) « Computation Tree Logic (en aglais) » permet d'exprimer des propriétés portant sur les arbres d'exécution (issus de l'état initial) du programme. Elle modélise des expressions du passé et du futur à partir d'un état du système. Une expressions CTL exprime des propriétés relatives à l'arbre d'exécution grâce une syntaxe et une sémantique.

2.3.1 Syntaxe de CTL

Le langage des formules de CTL sont les formules d'état définies inductivement par :

— $\forall p \in$ AP, p est une formule d'état ;

— Si p, q sont des formules d'état, alors $p \vee q$ et $\neg p$ sont des formules d'état ;

— Si p est une formule de chemin, alors E_p et Ap sont des formules de chemin ;

— Si p et q sont des formules d'état, alors X_p et pUq sont des formules d'état.

2.3.2 Sémantique de CTL

La sémantique de CTL est interprétée dans une structure de Kripke. Les états sont décorés par les propositions atomiques p.
Soit q_0 un état de S. La sémantique de CTL est définie inductivement par :

— $p \in$ AP, $q_0 \models p \Leftrightarrow p \in \mathrm{L}(q_0)$

— $q_0 \models p \vee q \Leftrightarrow q_0 \models p \; ou \; q_0 \models q$

— $q_0 \models \neg p \Leftrightarrow not(q_0 \models p)$

— $q_0 \models \mathrm{E}p \Leftrightarrow \exists \delta = q_0 \ldots q_n$ telle que $\delta \models p$ (p formules de chemins)

— $q_0 \models \mathrm{A}p \Leftrightarrow \forall \delta = q_0 \ldots q_n$ on a $\delta \models p$ (p formules de chemins)

— $\delta \models \mathrm{X}p \Leftrightarrow \delta_1 \models p$ (p formules d'état)

— $\delta \models p\mathrm{U}q \Leftrightarrow \exists j \geq 0, \delta_j \models q \; et (\forall k < j, \delta_k \models p)$ (p et q formules d'état)

les opérateurs temporels sont définies par :

— A (All), E (Exist) : quantifications sur toutes les exécutions possibles à partir de l'état courant

— X (next), F (eventually), G (always),U précédent directement A, E

— $\neg, \vee, \wedge$

Pour illustrer l'utilisation de la logique temporelle arborescence dans la formalisation des problèmes pratiques, nous donnons l'exemple du publiphone qui consiste à modéliser les actions qu'un utilisateur exécute pendant l'opération de tentative de téléphoner jusqu'à la fin de la communication. La procédure téléphonée est la suivante :

— L'utilisateur doit décrocher le téléphone ;

— Le système affiche insérer une carte ;

— L'utilisateur insère une carte ;

— Le système affiche le nombre d'unités restantes ;

— L'utilisateur compose son numéro de téléphone avec la possibilité de le corriger pour obtenir le bon numéro dans un délai de 2 secondes ;

— L'utilisateur communique avec son correspondant, tant qu'il lui reste des unités. – Si la carte est épuisée, l'utilisateur a 10 secondes pour la changer ;

— Lorsque la communication est terminée, l'utilisateur raccroche le téléphone ;

— Le système affiche retirer la carte. – L'utilisateur retire sa carte ;

Ensuite, formalisons les propositions atomiques :

— u_décrocher : l'utilisateur décroche le téléphone

— sys_affiche(message) : le système affiche le message

— u_insérer_carte : l'utilisateur insère une carte

— u_compser_no : l'utilisateur compose le numéros

— u_correction_no : l'utilisateur corrige le numéros

— numéros_correct : le numéros est correct

— u_communiquer : l'utilisateur communique avec son correspondant

— u_changer_carte_vide : l'utilisateur change sa carte qui est épuisée

— u_raccrocher : l'utilisateur raccroche le téléphone

— u_retirer_carte : l'utilisateur retire sa carte

— ctl_téléphoner : établit la clause de l'activité téléphoner

Enfin, on obtient la formalisation en CTL :

$$u\_decrocher \wedge u\_inserer\_carte \wedge$$
$$\mathrm{AX}[(u\_composer\_no \wedge \mathrm{EF}(u\_reprise\_sur\_erreur)) \wedge$$
$$\mathrm{AX}((u\_communiquer \wedge \mathrm{EF}(changer\_carte\_vide))\mathrm{U}$$
$$(u\_raccrocher \wedge u\_retirer\_carte))] \Rightarrow ctl\_telephoner$$

2.4 Model checking

Le model checking peut-être ramené à la vérification d'une propriété P sur un système $\phi : \phi \models P$. Plusieurs types de propriétés peuvent être vérifiés tels que les propriétés de sûreté ou de vivacité. Toutefois, vérifier des propriétés de sûreté est très souvent suffisant : On peut détecter des interblocages, vérifier des bornes ou encore qu'une section critique est bien respectée par les processus y accédant.

Vérifier une propriété de sûreté revient à parcourir les états que peut atteindre un système, et pour chaque état rencontré, vérifier si la propriété est respectée. Il s'agit donc d'un parcours de graphe (en largeur ou en profondeur).

Ainsi, la génération distribuée de l'espace d'états permet de répartir l'ensemble des états entre les machines du réseaux, ce qui permet de pallier au problème de l'explosion combinatoire de l'espace d'états. Le but de cette génération distribuée est la vérification de systèmes de tailles importantes. De ce fait, les algorithmes de vérification feront aussi l'objet de distribution.

L'algorithme de model checking distribué qui a été développé dans [ABIDINE, 2011], elle représente un version simplifiée de l'algorithme de model checking distribué. Toutes les machines contribuent pour la vérification des propriétés exprimées en CTL. Chaque machine vérifiée la propriété sur la structure de Kripke partiale, la vérification au niveau des états à parties inconnues est traité indécidable. Ainsi, les machines coopèrent afin de vérifier la formule CTL.

Dans cette section, nous présentons L'algorithme de model checking distribué qui a été proposer dans [ABIDINE, 2011].

2.4.1 Model checking CTL sur les fargments

Soit $M = (S, L, R, I)$ un fragment de structure de Kripke et une formule de CTL, l'algorithme récursif suivant calcule l'ensemble des états $H(f) \subseteq S$ qui satisfont f ou elle peut satisfaire f et qui exclut tous les états qui ne satisfont pas f.

$$T(p) = \{x \in S \mid (x,p) = true\} \tag{2.1}$$
$$U(p) = \{x \in S \mid (x,p) = \bot\} \tag{2.2}$$
$$F(p) = \{x \in S \mid (x,p) = false\} \tag{2.3}$$
$$pour\ x \in T, inT(x) = (1,x) \tag{2.4}$$
$$T^+(p) = \{inT(x) \mid \forall x \in T\} \tag{2.5}$$
$$pour\ x \in U, inU(x) = (-1,x) \tag{2.6}$$
$$U^+(p) = \{inU(x) \mid \forall x \in U\} \tag{2.7}$$
$$pour\ x \in F, inF(x) = (0,x) \tag{2.8}$$
$$F^+(p) = \{inF(x) \mid \forall x \in F\} \tag{2.9}$$
$$S^+(p) = T^+(p) \cup U^+(p) \cup F^+(p) \tag{2.10}$$
$$H(p) = T^+(p) \cup U^+(p);\ p\ une\ proposition\ atomique \tag{2.11}$$
$$H(\neq f) = \{inT(x) \mid x \in (S - map(snd, H(f))) \cup U^+(f)\} \tag{2.12}$$
$$H(f \wedge g) = H(f) \sqcup H(g) \tag{2.13}$$
$$H(f \vee g) = H(f) \sqcap H(g) \tag{2.14}$$
$$H(AXf) = \begin{pmatrix} \{inT(snd(e)) \mid \forall\, e \in S^+(f).succ(e) \subseteq T^+(f) \subseteq H(f)\} \cup \\ \left\{ \begin{array}{c} inU(snd(e)) \mid \forall\, e \in S^+(f).succ(e) \subseteq (T^+(f) \cup \\ U^+(f))\ and \exists\, e' \in succ(e)\ telque\ e'^+(f) \end{array} \right\} \end{pmatrix} \sqcup \{inU(x) \mid x \in border(M)\} \tag{2.15}$$

$$\mathrm{H}(\mathrm{EX}f) = \left(\begin{array}{c} \{in\mathrm{T}(snd(e)) \mid \forall\, e' \in \mathrm{S}^+(f).succ(e) \subseteq \mathrm{T}^+(f)\} \cup \\ \{in\mathrm{U}(snd(e)) \mid \forall\, e' \in \mathrm{S}^+(f).succ(e) \subseteq \mathrm{U}^+(f) \subseteq \mathrm{H}(f)\} \end{array} \right) \sqcup \{in\mathrm{U}(x) \mid x \in border(\mathrm{M})\} \tag{2.16}$$

$$\mathrm{H}(\mathrm{AG}f) = \nu\mathrm{Z}.((\mathrm{H}(f) \sqcap \mathrm{AXZ}) \tag{2.17}$$

$$\mathrm{H}(\mathrm{AF}f) = \mu\mathrm{Z}.(\mathrm{H}(f) \sqcup \mathrm{AXZ}) \tag{2.18}$$

$$\mathrm{H}(\mathrm{A}(f \cup g) = \mu\mathrm{Z}.(\mathrm{H}(g) \sqcup \mathrm{H}(f)) \tag{2.19}$$

Après l'application de l'algorithme récursif ci-dessus, nous avons $\forall s \notin \mathrm{H}(f) \Rightarrow \mathrm{L}(s,f) = false$. Les autres opérateurs, comme par exemple EG peuvent être déduites de l'ensemble des opérateurs cités ci-dessus, par exemple $\mathrm{H}(\mathrm{EG}f) = \mathrm{H}(\neg(\mathrm{AF}\neg f))$.

2.4.2 Model checking CTL distribué

l'algorithme model checking distribué considère l'ensemble de la formule $\phi \in \{f\mathrm{EG}f, \mathrm{AG}f, \mathrm{AF}f, \mathrm{EF}f, \mathrm{A}(f\mathrm{U}g), \mathrm{E}(f\mathrm{U}g)\}$ peut-être vérifiée dans les états border et aussi la vérité sur les états border est le paramètre passé au transformateur de prédicat comme suit :

$$\mathrm{Y} = \{s \in border(\mathrm{M}) \mid \mathrm{L}(s,\phi) = \mathrm{T}rueou\mathrm{L}(s,\phi) = \bot\} \phi\ est\ une\ formule \tag{2.20}$$

$$\mathrm{AF}p = \lambda\mathrm{Y}.\mu\mathrm{Z}.(p \vee \mathrm{Y} \vee \mathrm{AXZ}) \tag{2.21}$$

$$\mathrm{AG}p = \lambda\mathrm{Y}.\mu\mathrm{Z}.(p \vee \mathrm{Y} \wedge \mathrm{AXZ}) \tag{2.22}$$

$$\mathrm{EG}p = \lambda\mathrm{Y}.\mu\mathrm{Z}.(p \vee \mathrm{Y} \wedge \mathrm{EXZ}) \tag{2.23}$$

$$\mathrm{A}(p \cup q) = \lambda\mathrm{Y}.\mu\mathrm{Z}.(q \vee \mathrm{Y} \vee (p \wedge \mathrm{AXZ})) \tag{2.24}$$

$$\mathrm{E}(p \cup q) = \lambda\mathrm{Y}.\mu\mathrm{Z}.(q \vee \mathrm{Y} \vee (p \wedge \mathrm{EXZ})) \tag{2.25}$$

(2.26)

(2.27)

Y représente l'absence d'information sur l'état border, grâce à l'information obtenue au niveau des états border, la validitée de la formule peut être conclus sur toute la structure de Kripke.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les réseaux de Petri ainsi que les systèmes de transitions étiquetées et aussi la structure de Kripke. Ensuite nous avons présenté la logique temporelle arborescente et le model cheching distribué qui exploite l'information au niveau des états du graph ainsi qu'il permet la vérification malgré l'insuffisance d'information résultante de la distribution de l'espace d'états sur plusieurs nœuds.

Deuxième partie

Contribution

Chapitre 3

ALGORITHME DE DISTRIBUTION D'ESPACE D'ÉTATS BASÉE SUR LE COMPORTEMENT DES SYSTÈMES

Sommaire

3.1 Introduction

Dans les travaux [SAIDOUNI, 2012], [TABIB, 2016], [BENSETIRA, 2017], nous avons constaté que les méthodes de génération des espaces d'états pour le model checking consiste qu'un grand espace d'états non structuré soit divisé en parties de tailles équilibrées, de telle sorte que peu de transitions lient les différentes parties, car chaque transition externe peut entraîner une surcharge de communication.

Après une étude des espaces d'états générés, nous avons constaté que la diminution de transitions liant les différentes parties peut entraîner un mauvais équilibrage de charge entre les machines du réseau. Outre l'équilibrage de charge, il peut entraîner une mauvaise qualité de distribution de l'espace d'états car lorsqu'une formule du model checking n'est pas vérifiée sur un état la diminution de transitions externes n'empêche pas une surcharge de communication. La qualité de distribution est estimée en fonction de la quantité de communication requise pour l'exécution des tâches distribuées [**?**]. Ainsi La réalisation de ces objectifs nécessiterait des informations supplémentaires sur l'espace d'états considéré et les communications engendrées.

Ces raisons nous ont amené à proposer une nouvelle politique de redistribution qui vise à analyser le comportement d'un système donné en générant son espaces d'états et en extrayant les informations pertinentes sur les états et leurs connexions (transitions internes et externes). Ensuite, redistribuer les états pertinents selon une certaines heuristiques basé sur la théorie de jeux afin d'optimiser les performances du système. les machines sont considérées comme des joueurs, Chaque machine cherche a optimisée ces performances en définissant une bonne localité pour un état pertinent tout en optimise l'équilibre de charge et la quantité de communication entre les machines.

Dans ce qui suit nous présentons la nouvelle politique de redistribution de l'espace d'états basée sur le comportement du système ainsi que les résultats obtenus.

3.2 Politique de distribution de l'espace d'états basée sur le comportement du système

La politique proposée *CDS :Comportemental Distribution States*, vise à optimiser la distribution de l'espace d'états ainsi le temps de vérification du model checking. Pour un système donné spécifier à partir d'un réseau de Petri, nous utilisons un algorithme de génération parallèle pour construire la structure de Kripke distribuée, en utilisant la fonction *MD5* pour le partitionnement de l'espace d'états entre les machines du réseau. Après cela, lors de l'exécution du model checking, chaque machine analyse son fragment (structure de Kripke partiale) et élabore certaines statistiques sur les états par rapport a la vérification. Les statistiques générés sur chaque état mesure la dépendance de l'état par rapport aux états de la machine local ou aux états des machines distantes. Une fois le processus de vérification du model checking est terminé, Le protocole de redistribution peut être lancer afin de calculer pour chaque état *Externe* et *solliciter* sur lesquels la formule n'est pas vérifiée l'ensembles des états à déplacer et à dupliquer. Après le calcul des ensembles on décide, sur la base de l'écart minimum et maximum des états a stocké sur la machine, si l'ensemble peut être déplacer ou rester dans une machine afin de minimiser la quantité de communication entre les machines avec une bonne distribution, ainsi les transitions externes peut être réduit. L'algorithme général est l'Algorithme 1 :

Algorithm 1: CDS

1 Générer la structure de Kripke distribuée à partir de la spécification d'un réseau de Petri;
2 Exécution du model checking et élaboration des statistiques sur chaque état;
3 Redistribuer des états, en optimisant les performances du système;

Dans les sections suivantes, nous détaillons les phases de cet algorithme.

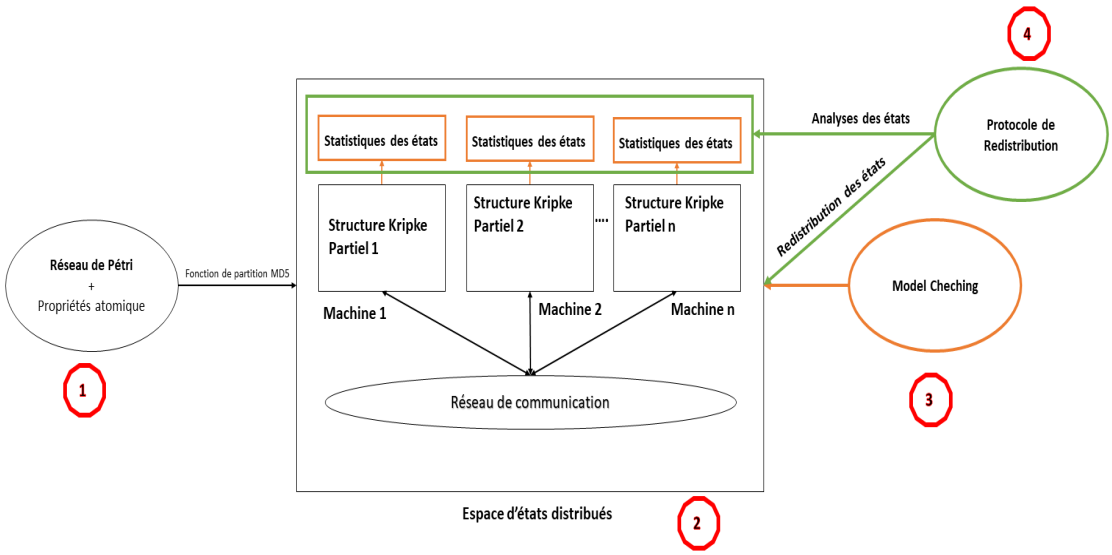


FIGURE 3.1 – Schéma de la politique de redistribution de l'espace d'états

3.2.1 Génération de l'espace d'états distribuée à partir de la spécification d'un réseau de Petri

La spécification du réseau de Petri est faite à partir d'une machine choisie aléatoirement. Le processus de génération de l'espace d'états distribuée est fait par l'exploration de l'état initial en générant tous ses états successeurs. Par la suite, toutes les machines disponibles sur le réseau contribuent à la construction des fragments de l'espace d'états distribué. Pour chaque nouvel état généré appartenant à une machine M_i, tous ses états successeurs sont générés. Un état successeur peut être dans la même machine ou dans une machine distante. Chaque machine M_i envoie tous ses états externes aux machines déterminées par la fonction de partition. La fonction de partition est basée sur la fonction de hachage cryptographique *MD5* qui renvoie un index j ($j \in 0, N-1$). La fonction de hachage adoptée réalise un bon équilibrage de charge entre les N machines du réseau. La génération distribuée se termine lorsqu'il n'y a aucun états en attente d'être exploré. L'algorithme de génération de l'espace d'états distribuée est l'Algorithme 2 :

Algorithm 2: Génération Initiale Distribuée

M_i : machine i
M_j : machine j
$Etat_i$: {dehors,dedans}
E_i : init à $\emptyset$, pile des états non encore explorés appartenant à la machine i
S_i : init à $\emptyset$, la liste des états déjà explorés appartenant à la machine i
s, m, m' : un état
$s.m$: marquage d'un états
s.L : liste des propriétés d'un états
TR_i : la liste des relations de transitions de la machine i
T : a liste de transitions
t_b : transition bloquant
$TExplore$: liste transition admissible
t : une transition
$Pres$: matrice pres du réseau de Petri
$Post$: matrice post du réseau de Petri
$Prop\_Pres$: matrice pres des propriétés
$Prop\_Post$: matrice post des propriétés

1 **begin**
2 **Reception Etat** (m :*etat*) *envoyé par* M_j
3 $s \leftarrow findByMarquage(m, S_i)$;
4 **if** ($s == null$) **then**
5 $m.sub \leftarrow \{M_j\}$;
6 Empiler(m,E_i);
7 **if** ($Etat_i$!= *dedans*) **then**
8 GenerationDistribue$_i$();
9 **else**
10 ajouter M_i à la liste des machines de l'etat portant identifiant m ;
11 **GenerationDistribue**$_i$ ()
12 $Etat_i \leftarrow dedans$;
13 **while** (E_i != $\emptyset$) **do**
14 $m \leftarrow depiler(E_i)$;
15 $TExplore \leftarrow \{t \in T \mid m.m[t>\}$;
16 $S_i \leftarrow S_i \cup \{m\}$;
17 **foreach** ($t \in TExplore$) **do**
18 $m'.m \leftarrow m.m + Post(t) - Pres(t)$;
19 $m'.L \leftarrow m.L + Prop\_Post(t) - Prop\_Pres(t)$;
20 $M_j \leftarrow MD5(m'.m)$;
21 **if** $M_j == M_i$ **then**
22 $s \leftarrow findByMarquage(m, S_i)$;
23 **if** (*findByMarquage(m'.m,*S_i*)==null and findByMarquage(m'.m,*E_i*)==null*) **then**
24 Empiler(m',E_i);
25 **else**
26 Envoyer (m') à M_j;
27 $TR_i \leftarrow TR_i \cup \{(m, t, m')\}$;
28 **if** (*TExplore*== $\emptyset$) **then**
29 $TR_i \leftarrow TR_i \cup \{(m, t_b, m)\}$;
30 $Etat_i \leftarrow dehors$;

```
1  Initialisation ()
2      T ← init;
3      Pres ← init;
4      Post ← init;
5      Prop_Pres ← init;
6      Prop_Post ← init;
7      m.m ← init;
8      m.prop ← init;
9      M_i ← MD5(m.m);
10     Envoyer (m) à M_j;
```

3.2.2 Principe Model Checking Distribué

Le model checking adopté est qui a été proposé par Bouneb Zine dans sa thèse [ABIDINE, 2011] (voir chapitre 2, section 2.4), il permet une vérification parallèles d'une formule φ sur les fragments de la structure de Kripke distribuée. Du fait de la distribution, la vérification de φ de certain état *S* dépend de la vérité en *S'* de cette formule, *S'* hébergé dans une machine distante M_j. La machine M_i effectue la vérification de φ sur le fragment détenue en considérant la valeur logique de *S'* indécidable $L(S', \varphi) = \perp$ car la formule φ peut être vérifier en *S'*. Lorsque la machine M_j termine le calcul est que la formule est vérifiée en *S'*, aucune notification n'est envoyée à la machine M_i, la machine M_i considère que la formule est vérifiée en *S'*, sinon la machine M_j envoi à la machine M_i la valeur logique de *S'* qui est $L(S', \varphi) = false$. Dans ce cas, M_i reprend le calcul de vérification avec la prise en compte de valeur logique envoyée afin de déterminer la logique de formule sur les états prédécesseurs. Une fois que la terminaison est détectée la machine détenant l'état initial déduit la véracitée de la formule φ.

Exemple 3.2.1. Soit une structure de Kripke distribuée représentée sur la Figure 3.2, l'exécution du model checking de la formule « AG(a) » sur cette structure, ainsi les résultat obtenue sont :

— Après une première itération l'algorithme détecte sur la machine M_1 que la formule n'est pas vérifiée sur l'état « S1 », cela nécessite l'envoyer d'une notification à la machine M_3 qui possède un prédécesseur direct de cet état.

— Après la réception d'une notification sur la machine M_3 et la prise en compte de la notification sur l'état « S1 », l'algorithme du model checking est relancé sur la machine M_3. Après le processus de vérification, la valeur logique de la formule sur l'état « S5 » est $false$ grâce à la deuxième itération. Après cela, aucune notification n'a été envoyé, sur la machine M_3 l'algorithme déduit alors la véracitée de la formule.

Les résultats observés durant le processus de vérification de la formule « AG(a) » sont représentés dans le Tableau 3.1, les lignes représentent les données concernant un état et les colonnes les types de données construits durant le processus de vérification. Les abréviations utilisées sont :

Id : L'identifiant de l'état ;

VL : La valeur logique de la formule à vérifier ($true : 1; false : 0; \perp : -1$) ;

Sub : Liste des machines sollicitant l'état ;

T : Le type d'état (I : interne ; E : externe) ;

Site : La machine distante où l'état (Externe) réside ;

I : La liste des itération permettant de déterminer la valeur logique de l'état.

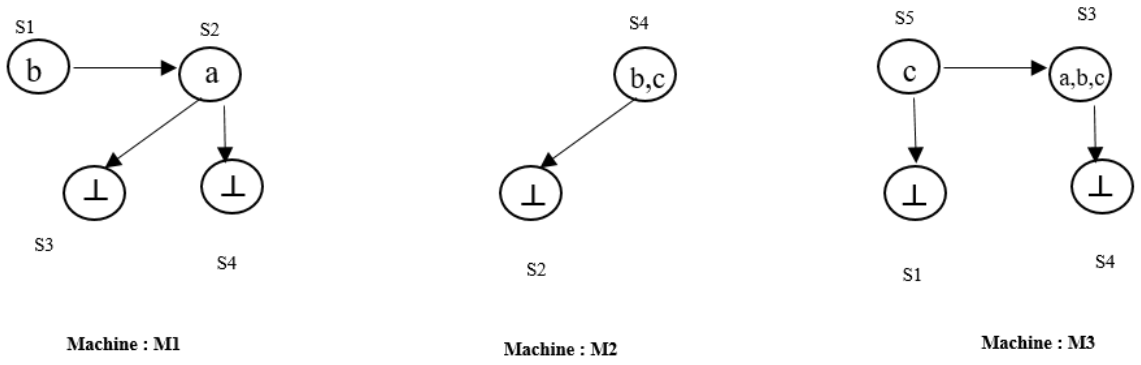


FIGURE 3.2 – Structure de kripke distribué

Itération 1						
Machine	Id	T	VL	I	Site	Sub
Machine M1	S1	I	0	1		$\{M_3\}$
	S2	I	-1	1		
	S3	E	-1	1	M3	
	S4	E	-1	1	M4	
Machine M2	S2	E	-1	1	M1	
	S4	I	-1	1		
Machine M3	S4	E	-1	1	M2	
	S5	I	-1	1		
	S3	I	-1	1		
	S1	E	-1	1	M1	
Itération 2						
Machine M3	S1	E	0	2	M1	
	S5	I	0	2		

TABLEAU 3.1 – Statistique des états

3.2.3 Paramètre d'optimisation

Après une analyse du principe précèdent nous avons constaté que la distribution des états liés entrainent un nombre de calculs assez élevé pour déterminer la valeur logique de la formule sur ces états. Le regroupement de ces états améliore le temps de traitement, cela nécessite la prise en considération des objectifs fixés. Ainsi, les paramètres α, β et γ sont utilisés pour atteinte ces objectif. La prise en compte simultanée des trois paramètres permet de minimiser à la fois le temps de traitement et assurer l'équilibrage de charge avec un nombre réduit de dupliquâts. La résolution de ce problème entraine l'utilisation de fonctions objectives qui permet de trouver l'espace de solution admissible car il peut existé une infinité de configurations respectant ces contraintes. Ainsi, différente techniques sont offertes pour l'obtention de l'espace des solutions, dans ce contexte la technique d'équilibre de Nash combinée aux fonctions objectives est envisageable.

α Le paramètre de temps de traitement ;

β Le paramètre d'équilibrage de charge entre les machines ;

γ Le paramètre de dupliquâts entre les machines.

3.2.4 Principe de l'équilibre de Nash

L'équilibre de Nash est une solution proposée par John Forbes Nash en 1950 [NASH, 1950] pour la recherche d'une solution optimale. Il est couramment utilisé en théorie des jeux. Un jeu présente une combinaison de décisions individuelles, appelées « stratégies », où chaque joueur anticipe correctement les choix des autres; Il y a autoréalisation, puisque l'issue réalisée est le fruit de décisions prises en pensant qu'elle va se réaliser. En théorie des jeux la question que se pose un joueur au moment de faire son choix est : que va faire l'autre? Ses croyances concernant le comportement des autres joueur ont donc un rôle essentiel au moment de la décision. La diversité de croyance correspond ainsi à une multiplicité d'équilibres. Dans les jeux coopératifs on autorise la communication et les accords entre joueurs avant la partie, les messages formulés par un joueur sont transmis sans modification à l'autre joueur, les accords entre joueurs seront respectés, ces hypothèses permettent d'obtenir un équilibre de Nash.

En utilisant le principe de Nash, J.A Désidéri [DESIDER, 2007] propose un algorithme de partitionnement de territoire qui se base sur des fonctions objectives. Il considère le cas où on dispose des fonctionnels prépondérants, ainsi l'algorithme cherche à optimiser les fonctions prioritaires avec une moindre dégradation tout en associant aux fonctionnelles secondaires des paramètres qui engendrent de grandes variations. La recherche de l'équilibre de Nash se fait par échange de résultats obtenus pour chaque fonction objective travaillant avec une partie seulement des variables, les autres étant fixées par les résultats obtenus pour les autres fonctions objectives. Cet équilibre est atteint quand l'optimisation de chaque fonction conduit toujours à la même solution.

En s'inspirant de ces principes, on propose une stratégie d'optimisation des paramètres définie ci-dessus.

3.2.5 Principe d'Optimisation de α, β *et* γ

Dans le cadre du model checking, les états présentent des informations par rapport à la vérification. L'analyse de ces résultats, montre que la distribution des états fortement liés augmente le temps de traitement de la vérification. A titre d'exemple sur l'exemple précédente, la distribution des états S1 et S5 a augmenté le temps de la vérification du model checking car ces états sont liés. Dans un exemple plus complexe où un ensemble d'états liés sont distribués, le temps de la vérification sera très élevé. Pour avoir une distribution de l'espace d'états respectant les objectifs fixés nous optimisons des paramètres α, β *et* γ) en simulant un jeu non coopératif entre les machines. L'optimisation de ces paramètres est fait comme suit :

— Sur Chaque machine on recherche le nombre d'états liés à chaque état sur lequel la formule n'est pa vérifiée, le paramètre β pour cet état est égale à ce nombre. Les états local qui disposent des prédécesseurs directs sur d'autres machines distantes, la valeurs du paramètre correspond aux nombres des successeurs directs et indirects liés au même états.

— La limite des états liés ou dépendant peut entraîner la duplication de certain états, ce nombre états correspond à la valeur du paramètre γ.

— En appliquant une heuristique sur les paramètres β *et* γ de deux machines qui sont liée par un état nous obtenons une optimisation du paramètre α.

Dans les sections suivantes, nous détaillons les phases de ce principe d'optimisation, la structure de Kripke présent sur la Figure 3.3 sera utilisé dans les exemples.

A partir du nombre d'états stockés dans machine il est possible de calculer entre deux machine le nombre minimum et maximum d'états susceptibles d'être stockés sur chaque machine. Le calcule de cette intervalle est comme suit :

— Soit ($\beta 1$ *et* $\gamma 1$) les valeurs de la machine M_1 respectivement pour la machine M_2 ($\beta 2$ *et* $\gamma 2$).

— $\overline{\beta}$ la moyenne des états $\overline{\beta} = \frac{\beta 1 + \beta 2}{2}$.

— L'écart type $\delta = \sqrt{\frac{\sum_{i=1}^{2} \beta_i^2}{2} - \overline{\beta}^2}$

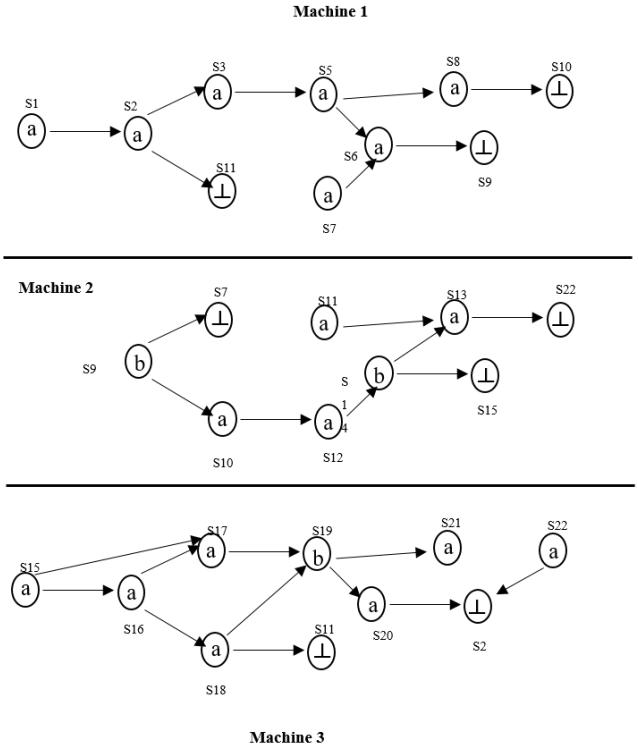


FIGURE 3.3 – Structure de kripke distribué

3.2.6 Calcul des valeurs des parametres β *et* γ

L'initialisation des paramètres s'effectue lors de l'exécution du model checking. Tel que nous l'avons souligné, ces paramètres sont initialisés à partir des états sur lesquels la formule n'est pas vérifiée. Par la suite, les états liés sont marqués par chaque état dont ils dépendent (dans l'algorithme c'est la variable **f** qui stocke cet ensemble d'états). Enfin, les états liés sont énumérés. Un état est dit lié, lorsque la formule n'est pas vérifiée sur ses successeurs directs ou indirects. Certains prédécesseurs directs ou indirects ne sont pas liés aux successeurs directs ou indirects sur lesquels la formule n'est pas vérifiée car celle-ci n'est pas vérifiée sur ces prédécesseurs. Cela motive la duplication de ces états prédécesseurs. A titre d'exemple, l'exécution de la formule AG(a) sur la structure de Kripke de la Figure(3.3), l'état « S9 » est un prédécesseur indirect de l'état « S14 », celui-ci peut être dupliqué sur une autre machine car la vérification de la formule sur cet état est indépendante des autres états.

Dans ce qui suit nous proposons la démarche d'initialisation des paramètres, le marquage des états liés, l'énumération des états liés et des états à dupliquer. Cette démarche est ajoutée à l'algorithme de model checking définit ci-dessus.

a Initialisation des paramètres β et γ

Les paramètres β et γ sont initialisés à zéro au niveau des états qui ne comportent pas la formule à vérifier. Lorsque l'état appartient à la machine, le paramètre γ sera initialisé à 1, cela signifie que cet état peut être dupliqué dans les machines qui le référencent. Par exemple, l'exécution de la formule AG(a) sur la structure de Kripke distribuée de la Figure(3.3) fait qu'à l'état « S9 », le paramètre γ est initialisé à 1 pour marquer la duplication de cet état. On remarque que la duplication de l'état « S9 » sur la « machine 1 », permet de réduire le temps de vérification de la formule, car la valeur logique de la formule sur à l'état « S9 » est connue, la « machine 2 » n'envoie pas alors de message concernant la valeur logique de la formule sur cet état. Le principe d'initialisation est formalisé sur l'équation (2.3). On obtient alors :

Algorithm 3: Initialize Parameters

```
S                : Liste des états
e                : Un état
p                : Une propriété
e.i              : Liste des itérations d'un état
L(e, p)          : Liste des propriétés d'un états
(type(e)         : Type d'un état : Border pour un état externe
curent_iteration : Itération courant
1 begin
2     foreach e in S do
3         if ( L(e, p) = false) and (e.i = null) ) then
4             e.β = 0 ;
5             e.γ = (type(e) ≠ Border)? 0 : 1;
6             e.i = {curent_iteration};
7         endif
8     end
9 end
```

Les instructions décrivant l'algorithme(4) sont expliquées comme suit :

ligne 2 : Parcourir l'espace d'états d'une « machine i ».

ligne 3 : Vérifier si la formule n'est pas vérifiée sur un état et si une itération précédente n'existe pas (c'est-à-dire si $e.i \neq null$ alors une itération l'avait déjà initialisée, mais il est à $null$.

Lorsque les conditions de la ligne 3 sont vérifiées alors les instructions des lignes suivantes sont exécutées :

ligne 4 : Le paramètre β est initialisé à 0(zéro).

ligne 5 : Le paramètre γ est initialisé à 0(zéro) lorsque l'état est un Border (c'est-à-dire lorsqu'il appartient à une autre machine) , sinon à 1. Par exemple, l'exécution de la formule AG(a) sur la structure de Kripke distribuée de la Figure(3.3).A l'état « S9 » : sur « machine 1 » le paramètre γ est initialisé à 0, car cet état n'appartient pas à la machine; Par contre sur la « machine 2 » le paramètre γ est initialisé à 1, car l'état appartient à cette machine.

ligne 6 : L'itération courante est sauvegardée sur l'état pour éviter de reprendre l'initialisation de cet état lors d'une prochaine itération car s'il n'est pas ajouté, les états respectant les conditions de la ligne 3 sont initialisés; sans cette initialisation, cet état vérifiera les conditions de la ligne 3.

Exemple 3.2.2. Après la vérification de la formule AG(a) sur la structure de Kripke de la Figure (3.3), les résultats d'initialisation sont :

Itération 1 Dans la première itération, le paramètre γ est initialisé à 1 sur tous les états sur lesquels la formule n'est pas vérifiée. L'attribution de la valeur 1 est expliquée au niveau de la ligne 5 de l'algorithme 4. Par contre le paramètre β est sans contrainte, alors il est à 0. Le numéro d'itération est enregistré sur ces états. La sauvegarde de l'itération courante est expliquée à la ligne 6 de l'algorithme 4. Dans le Tableau (3.2), on retrouve les états initialisés par les machines.

Id	β	γ	I	M	T
S9	0	1	1	M2	I
S14	0	1	1	M2	I
S19	0	1	1	M3	I

TABLEAU 3.2 – Étape d'initialisation itération 1

Itération 2 Après la première itération, les états sur lesquels la formule n'est pas vérifiée, et qui ont des prédécesseurs appartenant à d'autres machines sont notifiées pour prendre en compte la valeur logique de ces états. Une notification déclenche une itération. Elle permet de vérifier la formule sur l'espace d'état de la machine notifiée. Pendant l'itération, l'algorithme du model checker détecte sur certains états que la formule n'est pas vérifiée. Ces états sont liés aux états sur lesquels la notification est faite. Une seconde itération est déclenchée sur la « machine 1 », car l'état « S6 » dispose d'un successeur sur la « machine 2 » (l'état « S9 »). La formule n'est pas vérifiée sur cet état(détecté à Itération 1), d'où la notification envoyée par la « machine 2 » à la « machine 1 ». Dans la « machine 1 » les paramètres (β, γ) de l'état « S9 » sont initialisés à zéro. Le paramètre γ est initialisé à 0 parce qu'il n'appartient pas à la « machine 2 ». Ainsi le Tableau (3.3) décrit les différents états initialisés dans les machines notifiées.

Id	β	γ	I	M	T
S9	0	0	2	M1	E
S15	0	0	2	M2	E
S10	0	0	2	M1	E

TABLEAU 3.3 – Étape d'initialisation itération 2

Itération 3 Après l'itération précédente, des états liés peuvent posséder des prédécesseurs sur d'autres machines. Ces dernières machines sont notifiées. Par exemple, l'état « S2 » est lié à l'état « S9 », « S2 » possède des prédécesseurs sur la « machine 2 » et sur la « machine 3 ». La notification pour cet état entraine une vérification de la formule sur chaque machine notifiée. Ainsi les paramètres(β, γ) de cet état sont initialisées à 0 d'après l'algorithme d'initialisation. Le paramètre γ est initialisé à 0 parce qu'il n'appartient pas à la machine. Le Tableau (3.4) décrit les différents états initialisés sur chacune des machines.

Id	β	γ	I	M	T
S7	0	0	3	M2	E
S2	0	0	3	M3	E

TABLEAU 3.4 – Étape d'initialisation itération 3

Itération 4 Après l'itération précédente, des états liés peuvent posséder des prédécesseurs sur d'autres machines. Ces dernières sont notifiées. Par exemple l'état « S22 » est lié à l'état « S2 ». Il possède un prédécesseur sur la « machine 2 ». La notification pour cet état, déclenche une vérification de la formule sur la « machine 2 ». Ainsi les paramètres(β, γ) de cet état sont initialisés à 0, d'après l'algorithme d'initialisation. Le paramètre γ est initialisé à 0 parce qu'il n'appartient pas à cette machine. Le Tableau (3.5) décrit les différents états initialisés sur chacune des machines.

Id	β	γ	I	M	T
S22	0	0	4	M2	E

TABLEAU 3.5 – Étape d'initialisation itération 4

Itération 5 Après l'itération précédente, des états liés peuvent posséder des prédécesseurs sur d'autres machines. Les machines sont notifiées. Par exemple l'état « S11 » est lié à l'état « S2 », des prédécesseurs sont sur la « machine 1 » et sur la « machine 3 ». La notification pour cet état, déclenche une vérification de la formule sur chaque cette machine. Les paramètres (β, γ) de cet état sont initialisés à 0, d'après l'algorithme d'initialisation. Le paramètre γ est initialisé à 0 parce qu'il n'appartient pas à cette machine. Le Tableau (3.6) décrit les différents états initialisés sur chacune des machines.

Id	β	γ	I	M	T
S11	0	0	5	M1	E
S11	0	0	5	M3	E

TABLEAU 3.6 – Étape d'initialisation itération 5

b Marquage des états liés

Pendant l'exécution de l'algorithme de vérification, la valeur logique de la formule sur certains états peut dépendre des successeurs directs ou indirects, ces états sont alors liés. La formule peut être vérifiée sur les successeurs, ils peuvent appartenir à d'autres machines. Lorsque la formule n'est pas vérifiée sur un état successeur, les états liés sont alors marqués par cet état. Le principe de marquage est implémenté sur les deux algorithmes de base du model checker. Ils sont définis comme suit :

Premier algorithme : Il est défini par l'équation (2.15), il est utilisé lorsque la vérification concerne tous les successeurs. Ainsi Lorsque la formule n'est pas vérifiée au niveau d'un successeur, deux cas se présentent :

— La propriété est vérifiée sur l'état, l'état est alors lié à un successeur. Les successeurs sur lesquels la formule n'est pas vérifiée sont marqués sur l'état.

— Lorsque la propriété n'est pas vérifiée sur un état, celui-ci peut être dupliqué soit sur la machine réalisant le traitement soit sur des machines distantes. Les états successeurs sur lesquels la formule n'est pas vérifiée sont enregistrés dans l'ensemble « limit », cela signifie que cet état est indépendant de ses successeurs et peut donc être dupliqué sur des machines distantes. Sur l'exemple (3.2.2) de l'Algorithme(4), la valeur logique de la formule à l'état « S9 » est indépendante des états « S14 » et « S7 ». L'état « S9 » peut alors être dupliqué sur les machines qui possèdent ses prédécesseurs.

Deuxième algorithme : Il est défini par l'équation(2.16). Il se différencie du premier algorithme par la vérification de la formule sur les successeurs. Il vérifie l'existence de la propriété sur un successeur.

En plus des conditions des algorithmes, l'existence de l'itération courante est vérifiée. Cette condition permet de marquer les états de cette itération. Ce principe est formalisé comme suit :

Algorithm 4: Mark linked states formula AX

```
1  AXf ⟵ ∅;
2  foreach e in S do
3      if ((∀e' in succ(e) ⊆ (T⁺(f) ∪ U⁺(f))) and (e ∈ {T⁺(f) ∪ U⁺(f)})) then
4          AXf ⟵ AXf ∪ {e};
5      else
6          if (e in T) then
7              foreach e' in succ(e) do
8                  if (curent_iteration in e'.i) then
9                      e.f ⟵ e.f ∪ e'.f;
10                     e.i ⟵ e.i ∪ {curent_iteration};
11                 endif
12             end
13         else
14             foreach e' in succ(e) do
15                 if (curent_iteration in e'.i) then
16                     e.limit ⟵ e.limit ∪ e'.f;
17                     e.fn ⟵ e.fn ∪ {curent_iteration};
18                 endif
19             end
20         endif
21     endif
22 end
```

Les instructions décrivant l'Algorithme(5) sont expliquées comme suit :

ligne 1 : L'ensemble « AXf » contient les états sur lesquels la formule est vérifiée ou peut être vérifiée. Cet ensemble est initialisé par l'instruction de cette ligne.

ligne 2 : L'instruction de cette ligne permet de parcourir les états stockés sur la « machine i »

ligne 3 : L'instruction de cette ligne vérifie la formule sur les successeurs, si elle est vérifiée, alors l'instruction de la ligne(4) est exécutée.

ligne 4 : L'instruction de cette ligne stocke l'état dans l'ensemble « AXf ».

ligne 5 : Lorsque la formule n'est pas vérifiée sur l'état ou ses successeurs, les instructions des lignes suivantes sont alors exécutées.

ligne 6 : L'instruction de cette ligne vérifie la formule sur l'état. Lorsqu'elle est vérifiée, les instruction des lignes(7 à 12) sont alors exécutées. La ligne 7 permet de parcourir les successeurs de l'état pour vérifier l'itération courante grâce à l'instruction de la ligne 8. Une fois qu'elle est présente, les instructions suivantes sont exécutées :

ligne 9 : L'instruction de cette ligne permet de récupérer les états dont ils dépendent.

ligne 10 : L'instruction de cette ligne enregistre l'itération courante sur l'état, car un état peut dépendre de plusieurs états de différentes itérations.

ligne 13 : Lorsque la formule n'est pas vérifiée sur l'état,ses successeurs sont parcourus grâce à l'instruction de la ligne 14. Lors de ce parcours, l'itération courante est vérifiée par l'instruction de la ligne 15.
Lorsque l'itération courante est présente, les états successeurs sur lesquels la formule n'est pas vérifiée sont enregistrés dans la variable « limit » de l'état grâce à l'instruction de la ligne 16. La sauvegarde de ces états permet d'éviter le déplacement de cet état, étant donné que sa duplication permet à l'algorithme de connaitre la valeur logique de ces prédécesseurs sans que la machine concernée ne soit notifiée.

Algorithm 5: Mark linked states formula EX

```
1  EXf ⟵ ∅;
2  foreach e in S do
3      if (∃e' in succ(e) ⊆ (T⁺(f) ∪ U⁺(f))) then
4          EXf ⟵ EXf ∪ {e};
5      else
6          if (e in T) then
7              foreach e' in succ(e) do
8                  if (curent_iteration in e'.i) then
9                      e.f ⟵ e.f ∪ e'.f;
10                     e.i ⟵ e.i ∪ {curent_iteration};
11                 endif
12             end
13         else
14             foreach e' in succ(e) do
15                 if (curent_iteration in e'.i) then
16                     e.limit ⟵ e.limit ∪ e'.f;
17                     e.fn ⟵ e.fn ∪ {curent_iteration};
18                 endif
19             end
20         endif
21     endif
22 end
```

Les instructions décrivant l'Algorithme(6) sont similaires à celle de l'Algorithme(5), la différence se trouve au niveau de la vérification de la formule sur les successeurs. Pour l'Algorithme(6), lorsque la formule est vérifiée ou peut être vérifiée sur un successeur, l'instruction de la ligne 4 est alors exécutée, sinon les instructions de la ligne 5 à la ligne 21 sont exécutées en fonction de la vérification de la formule comme expliqué au niveau de l'Algorithme(5).

Exemple 3.2.3. L'application de l'algorithme de model checker de la formule AG(a) sur la structure de Kripke de la Figure (3.3) donne les états marqués à chaque itération comme

suit :

Itération 1 Dans la première itération, les états initialisés dans l'exemple(3.2.2) sont utilisés pour déterminer les états liés. Le model checker utilise l'Algorithme(5) pour vérifier la formule AG(a), cet algorithme détecte sur les états(« S12 »,« S10 ») qu'ils sont liés à l'état « S14 », car la formule n'est pas vérifiée sur l'état « S14 », par contre la formule est vérifiée sur ces états. La vérification de la formule sur les successeurs directs de l'état « S14 », ce dernier est un successeur direct de « S12 », ou indirect s de l'état « S14 », ce dernier est un successeur indirect de l'état « S10 », cela montre qu'elle n'est pas vérifiée sur chacun de ces états. l'état « S14 » est marqué dans l'ensemble de dépendance des états liés. L'état « S9 » est un prédécesseur indirect de l'état « S14 », ce dernier est marqué sur l'ensemble « limit » de l'état « S9 » car la formule n'est pas vérifiée sur état « S9 ». Le Tableau (3.7) représente les différents états marqués par chacune des machines pendant cette itération.

Id	T	F	I	limit	fn	M
S9	I	∅	{1}	{S14}	{S14}	M2
S10	I	{S14}	{1}	∅	∅	M2
S12	I	{S14}	{1}	∅	∅	M2
S15	I	{S19}	{1}	∅	∅	M3
S16	I	{S19}	{1}	∅	∅	M3
S17	I	{S19}	{1}	∅	∅	M3
S18	I	{S19}	{1}	∅	∅	M3

TABLEAU 3.7 – Étape de marquage : itération 1

Itération 2 Dans cette itération les résultats obtenues dans l'exemple(3.2.2) à l'itération 2 sont utilisés pour déterminer les états liés aux états notifiés(dans l'exemple(3.2.2) une notification provoque l'itération 2 sur la « machine 1 », cette notification concerne l'état « S9 »). Pendant la vérification de la formule, l'algorithme détecte la valeur logique de la formule au niveau des états(« S1 », « S2 », « S3 », « S4 »,« S5 », « S6 ») sur la « machine 1 ». La valeur logique de cette formule sur ces états dépend de la valeur logique de la formule sur l'état « S9 ». Ces états sont des prédécesseurs directs ou indirects de l'état « S9 ». Cet état est rajouté à l'ensemble de dépendance, l'itération courante est ainsi marquée. Le Tableau (3.8) représente les différents états marqués pendant l'itération 2 de chacune des machines.

Itération 3 Dans cette itération les résultats obtenues dans l'exemple(3.2.2) à l'itération 3 sont utilisés pour déterminer les états liés aux états notifiés de l'exemple(3.2.2. Une notification provoque l'itération 3 sur la « machine 3 », cette notification concerne l'état « S2 »). Pendant la vérification, l'algorithme détecte la valeur logique de l'état(« S20 ») sur la « machine 3 », elle dépend de l'état « S2 ». Ce dernier est ajouté à l'ensemble de dépendance de l'état « S20 ». Par contre l'état « S19 » est un prédécesseur indirect de l'état « S2 », il est ajouté à l'ensemble « limit » de l'état (« S19 », car la formule n'est pas vérifiée sur cet état l'à. Toute fois l'itération courante est marquée sur l'état « S19 ». le Tableau (3.9) représente les différents états marqués pendant l'itération 3 de chacune des machines.

Id	T	F	I	limit	fn	M
S1	I	{S9,S10}	{1,2}	∅	∅	M1
S2	I	{S9,S10}	{1,2}	∅	∅	M1
S3	I	{S9,S10}	{1,2}	∅	∅	M1
S5	I	{S9,S10}	{1,2}	∅	∅	M1
S6	I	{S9}	{2}	∅	∅	M1
S7	I	{S9}	{2}	∅	∅	M1
S8	I	{S10}	{2}	∅	∅	M1
S9	E	∅	{2}	∅	∅	M1
S10	E	∅	{2}	∅	∅	M1
S14	I	∅	{1}	{S15}	{S15}	M2
S15	E	∅	{2}	∅	∅	M2

TABLEAU 3.8 – Étape de marquage : itération 2

Id	T	F	I	limit	fn	M
S9	I	∅	{1}	{S7}	{S7}	M2
S7	E	∅	{3}	∅	∅	M2
S2	E	∅	{3}	∅	∅	M3
S19	I	{S19}	{1}	{S2}	{S2}	M3
S20	I	{S2}	{3}	∅	∅	M3
S22	I	{S2}	{3}	∅	∅	M3

TABLEAU 3.9 – Étape de marquage : itération 3

Itération 4 Dans cette itération les résultats obtenues dans l'exemple(3.2.2) à l'itération 4 sont utilisés pour déterminer les états liés aux états notifiés dans l'exemple(3.2.2. Une notification provoque l'itération 4 sur la « machine 2 », la notification concerne l'état « S22 »). Le principe de marquage est similaire à l'itération précédente, le Tableau (3.10) représente alors les différents états marqués pendant l'itération 4 de chacune des machines.

Id	T	F	I	limit	fn	M
S11	I	{S22}	{4}	∅	∅	M2
S13	I	{S22}	{4}	∅	∅	M2
S14	I	∅	∅	{S15,S22}	{S15,S22}	M2
S22	E	∅	{4}	∅	∅	M2

TABLEAU 3.10 – Étape de marquage : itération 4

Itération 5 Dans cette itération les résultats obtenues dans l'exemple(3.2.2) à l'itération 5 sont utilisés pour déterminer les états liés aux états notifiés de l'exemple(3.2.2. Une notification provoque l'itération 5 sur la « machine 3 », la notification concerne l'état « 11 »). Ainsi l'algorithme détecte la valeur logique des états(« S15 », « S16 », « S18 ») sur la « machine 3 », elle dépend de l'état « S11 ». Le principe de marquage est similaire à l'itération 2, on remarque sur ces états l'effet de plusieurs itérations, cela

s'explique par le fait que ces états dépendent de plusieurs autres états. Le Tableau (3.11) représente les différents états marqués pendant l'itération 5 de chacune des machines.

Id	T	F	I	limit	fn	M
S1	I	∅	{1,5}	{S7}	{S7}	M2
S2	I	∅	{1,5}	∅	∅	M2
S11	I	∅	{5}	∅	∅	M2
S11	E	∅	{5}	∅	∅	M3
S15	I	{S19, $s11$}	{1,5}	∅	∅	M3
S16	I	{S19, $s11$}	{1,5}	∅	∅	M3
S18	I	{S19, $s11$}	{1,5}	∅	∅	M3

TABLEAU 3.11 – Étape de marquage : itération 5

c Énumération des états liés

Après le marquage des états, on obtient sur les états liés les états dont ils dépendent. Ainsi pour chaque état dépendant, les états qui lui sont liés sont énumérés, cela permet d'attribuer des valeurs aux paramètres (β et γ). L'attribution des valeurs peut concerner les états appartenant à la machine locale ou appartenant aux machines distantes. Ces valeurs permettent d'effectuer un redéploiement des états sur la bonne machine. Selon le type de chaque état la technique d'énumération est la suivante :

Etat Externe : Les états externes(« border ») sont des états appartenant aux machines distantes. Lorsque la formule n'est pas vérifiée sur ces états, le nombre d'états liés sont recensés. La valeur calculée est attribuée au paramètre β de l'état « border ». Dans l'exemple(3.2.3), la valeur du paramètre β est égale à 2 à l'état « S22 » de la « machine 2 ». Ainsi la valeur du paramètre γ correspond au nombre d'états pour lesquels l'état « border » a été inséré dans leurs ensembles respectifs « limit ». Dans le cas des états externes, la duplication de ces états sur une machine donne une structure de Kripke incohérente (la structure obtenue présente des états dupliqués sans aucuns prédécesseurs sur la machine elle même, ce modèle est alors incohérent vis à vis d'une structure de Kripke). Pour obtenir une structure cohérente il est envisageable de déplacer ces états sur d'autres machines tout en laissant les duplicatas sur la machine local. Pour ce faire le paramètreγm est utilisé pour stocker le nombres de ces états à la place du paramètre γ. La valeur de ce paramètre est calculée similairement à celle du paramètre γ. L'application de ce principe sur l'exemple(3.2.3) permet d'attribuer la valeur 1 au paramètre γm à l'état « S22 » de la « machine 2 », car cet état est stocké dans l'ensemble « limit » de l'état « S14 ».

Etat Interne : Les états internes sont des états appartenant à la machine locale. Lorsque ces états disposent des prédécesseurs directs sur d'autres machines distantes, les valeurs des paramètres sont alors calculés. La valeur du paramètre β correspond aux nombres des successeurs directs et indirects présents entre les état marqués sur l'ensemble de dépendance (la variable « **f** ») de l'état, ce nombre est incrémenté de 1 car l'état dépend de ces états l'à. Par contre la valeur du paramètre γ correspond aux nombre des états internes présents sur l'ensemble de dépendance. L'application de ce principe sur l'exemple(3.2.3) permet d'obtenir la valeur 2 pour le paramètre β et 0 pour le paramètre β à l'état « S7 » de la « machine 2 ».

Ces principes sont formalisés comme suit :

Function 1 getSucess($e : state$) : List of State

$ElementSucc$**:** Liste des états Successeur
e **:** Un état
$e.f$ **:** Liste que dépendant un état
$succ(e)$ **:** Successeur d'un état : Border pour un état externe

```
1  begin
2      ElementSucc ⟵ ∅  foreach e' ∈ succ(e) do
3          if ((e' ♯ e.f) and ((e'.f ∩ e.f) ≠ ∅) then
4              ElementSucc ⟵ ElementSucc ∪ {e'} ∪ (e')
5      return ElementSucc
```

Les instructions de la fonction « getSucc » permettent de récupérer les successeurs directs et indirects entre un état et les états dont-il dépend. La ligne 2 permet de parcourir les successeurs directs de l'état afin de vérifier leur dépendance avec lui. Cela est faite à la ligne 3. Lorsque le successeur est en dépendance avec l'état encours, il est ajouté à l'ensemble des successeurs suivi de l'appel de la fonction « getSucc » pour le traitement transitif de ses successeurs. Le résultat retourné par cette fonction est ajouté à l'ensemble des successeurs. Ces instructions sont exécutées à la ligne 4. Ainsi, la fonction retournera touts les successeurs de l'état en cours à la fin de son exécution.

Algorithm 6: Count parameters values

```
1  begin
2      foreach e in S do
3          if ((e ∈ border(M)) and (e ∈ F) and (e.i == curent_iteration)) then
4              foreach e' in S − {e} do
5                  if ((e ∈ e.f)) then
6                      e.β ⟵ e.β + 1;
7                  else if (e ∈ e'.limit) then
8                      e.β ⟵ e.β + 1;
9                      e.γm ⟵ e.γm + 1;
10             end
11         else if ((e ∈ Notifier(M)) and (e ∈ T) and (curent_iteration ∈ e.i)) then
12             e.β ⟵ Count(getSucc(e)) + 1;
13             foreach e" ∈ e.f do
14                 if ((Type(e") ≠ BorderANDe"! = e) then
15                     e.γ ⟵ e.γ + 1;
16                 endif
17             end
18     end
19 end
```

Les instructions décrivant l'Algorithme **??** sont expliquées comme suit :

ligne 2 : Cette ligne présente une instruction de parcours des états stockés sur la « machine i ».

ligne 3 : L'instruction de cette ligne vérifie que l'état est « externe », la formule n'est pas vérifiée à cet état, et l'itération courante est marquée sur l'état. La dernière condition permet de calculer les valeurs des paramètres d'un état de l'itération courante vérifiant les deux premières conditions. Lorsque ces conditions sont vérifiées, l'espace d'états est alors parcouru par l'instruction de la ligne 4, et calcule les valeurs des paramètres. Pour chaque état est vérifiée :

— Si l'état externe est présent dans l'ensemble de dépendance de l'état encours, le paramètre β de l'état externe est alors incrémenté (ligne 6) car cet état est lié à l'état externe.

— Sinon, si l'état externe est présent dans l'ensemble « limit » de cet état, les paramètres β et γm sont alors incrémentés (ligne 8 et ligne 9) car cet état peut être déplacer en laissant un dupliquât sur la machine effectuant le traitement.

ligne 11 : L'instruction de cette ligne vérifie sur l'état l'existence des prédécesseurs directs sur des autres machines distantes, la dépendance des successeurs et la présence de l'itération courante. Lorsque ces conditions sont vérifiées, les successeurs entre les états liés sont alors calculés, car la valeur du paramètre β correspond au nombre des successeurs entre l'état et les états dont-il dépend (ligne **??**). Par contre la valeur du paramètre γ correspond au nombre des états dont-il dépend appartenant à la machine locale, les instructions sont de la ligne 13 jusqu'à la ligne 17.

Exemple 3.2.4. L'application de l'Algorithme(**??**) permet de déterminer pour chacun des états de l'exemple(3.2.2), les valeurs des paramètres à travers les résultats de l'exemple(3.2.3). Les valeurs calculées sont :

Itération 1 La première itération comptabilise pour chaque état « Border » ou « Notifer » de l'exemple(3.2.2) à l'itération 1 le nombre des états qui lui sont liés. Ce nombre est affecté au paramètre β, par exemple dans l'exemple(3.2.3) aucun état dépend de l'état « S9 » sur la « machine 2 ». Cependant, il peut-être dupliqué sur la « machine 1 » car la formule n'est pas vérifiée sur cet état, la valeur du paramètre γ est alors égale à 1. Ainsi, les valeurs des paramètres de l'état « S14 » ne sont pas calculés car il ne possède pas de prédécesseurs sur chacune des autres machines. Le Tableau (3.12) représente les valeurs des paramètres des états.

Id	β	γ	γm	I	M	T
S9	0	1	0	1	M2	Notifier
S10	2	1	0	1	M2	Notifier
S15	3	1	0	1	M3	Notifier

TABLEAU 3.12 – Calcul des valeurs des parametres : itération 1

Itération 2 Dans cette itération les valeurs de l'état « S15 » sont expliquées car le calcul des paramètres des autres états est similaire à l'itération précédente. Ainsi, dans l'exemple(3.2.3) il n'existe pas d'état dépendant de l'état « S15 » sur la « machine 2 », par contre sur l'état « S14 » il est inséré dans l'ensemble « limit ». Ainsi les valeurs des paramètres β et γm sont égales à 1 car l'état « S14 » peut être déplacé sur la « machine 2 ». Cet état est donc dupliqué sur la machine locale. Le Tableau (3.13) représente les valeurs des paramètres des états.

Itération 3 Dans cette itération, la démarche pour l'état « S2 » est similaire à celle de l'état « S15 » auquel est associé la démarche de l'itération 1 pour calculer les états liés à

Id	β	γ	γm	I	M	T
S2	6	0		2	M1	Notifier
S7	2	0		2	M1	Notifier
S9	6	0		2	M1	Border
S15	1	0	1	2	M2	Border

TABLEAU 3.13 – Calcul des valeurs des paramètres : itération 2

Id	β	γ	γm	I	M	T
S2	3	0	1	3	M3	Border
S7	1	0		3	M2	Border
S22	1	0		3	M3	Notifier

TABLEAU 3.14 – Calcul des valeurs des paramètres : itération 3

cet état. Par contre les autres états sont similaires à l'itération 1. Le Tableau (3.14) représente les valeurs des paramètres des états.

Itération 4 La démarche de cette itération est similaire à l'itération précédente. Le Tableau (3.15) représente les valeurs des paramètres des états.

Id	β	γ	γm	I	M	T
S11	2	0	0	4	M2	Notifier
S2	3	0	1	4	M2	Border

TABLEAU 3.15 – Calcul des valeurs des paramètres : itération 4

Itération 5 La démarche de cette itération est similaire à l'itération 3. Le Tableau (3.16) représente les valeurs des paramètres des états.

d Échange des valeurs des paramètres

Après le calcul des valeurs des paramètres, ces valeurs sont envoyées aux machines concernées par l'état. A la réception des valeurs, ces dernières sont stockées sur l'état concerné. Ce prince est formalisé comme suit :

L'envoi des valeurs concernent les états appartenant aux machines distantes sur lesquels la formule n'est pas vérifiée, et les états locale la formule n'est pas vérifiée qui possèdent des prédécesseurs directs sur d'autres machines. Le parcours de ces états est fait par l'instruction de la ligne 1 de la procédure d'envoi. L'envoi des valeurs concerne les états de l'itération courante, car les valeurs sont échangées après chaque itération. La vérification de la présence de l'itération courante est effectuée à la ligne 2. Lorsqu'elle est pressente, les valeurs calculées sont envoyées à toutes les machines concernées par cet état. Cela est fait de la ligne 3 jusqu'à la ligne 6.

Exemple 3.2.5. Le processus d'envoi des valeurs obtenues dans l'exemple(3.2.4), celuici montre que les valeurs de certains états ne sont pas envoyées car ces états ne disposent pas de prédécesseurs directs sur des machines distantes. Les valeurs envoyées pour chaque état sont décrites comme suit :

Itération 1 — La « machine 2 » envoie les informations : id =s9 ;β=0 ;γ=1 ;γm=0 ;f ={S9} ;i=2(le paramètre i désigne l'identité de la machine) à la « machine 1 ».

Id	β	γ	γm	I	M	T
S11	2	0	0	5	M1	Border
S11	3	0	0	5	M3	Border
S15	4	1	0	5	M3	Notifier

TABLEAU 3.16 – Calcul des valeurs des paramètres : itération 5

Procedure 1 SendValueCalculate $(id, \beta, \gamma, \gamma m, f)$

begin

1 **foreach** $e \in \{Notifier(S) \cup Border(S)\}$ **do**
2 **if** $(curent_{iteration} \in e.i)$ **then**
3 F = $(type(e) == Notifier)?e : \emptyset$
4 **foreach** $m \in e.site$ **do**
5 $send(e.id, e.\beta, e.\gamma, F, e.\gamma m, i)$ to $m.id)$

— La « machine 2 » envoie les informations : id =s10 ;β=2 ;γ=1 ;γm=1 ;f ={S10} ;i=2 à la « machine 1 ».

— La « machine 3 » envoie les informations : id =s15 ;β=3 ;γ=0 ;γm=1 ;f ={s19} ;i=3 à la « machine 2 ».

Itération 2 — La « machine 1 » envoie les informations : id =s9 ;β =6 ;γ =0 ;γm=0 ;f =Ø, i=1à la « machine 2 ».

— La « machine 1 » envoie les informations : id =s10 ;β=5 ;γ=0 ;γm=0 ;f =∅ ;i=1 à la « machine 2 ».

— La « machine 1 » envoie les informations : id =s7 ;β=2 ;γ=0 ;γm=0 ;f ={s9} ;i=1 à la « machine 2 ».

— La « machine 1 » envoie les informations : id =s2 ;β=5 ;γ=0 ;γm=0 ;f ={s9, s10} ;i=1 à la « machine 3 ».

— La « machine 2 » envoie les informations : id =s15 ;β=1 ;γ=0 ;γm=1 ;f =∅ ;i=2 à la « machine 3 ».

Itération 3 — La « machine 2 » envoie les informations : id =s7 ;β=1 ;γ=0 ;γm=1 ;f =∅ ;i=1 à la « machine 1 ».

— La « machine 3 » envoie les informations : id =s2 ;β=3 ;γ=0 ;γm=1 ;f =∅ ;i=3 à la « machine 1 ».

— La « machine 3 » envoie les informations : id =s22 ;β=1 ;γ=0 ;γm=0 ;f ={s2} ;i=3 à la « machine 2 ».

Itération 4 — La « machine 2 » envoie les informations : id =s11 ;β=2 ;γ=0 ;γm=0 ;f ={s22} ;i=2 à la « machine 1 ».

— La « machine 2 » envoie les informations : id =s11 ;β=2 ;γ=0 ;γm=0 ;f ={s22} ;i=2 à la « machine 3 ».

— La « machine 2 » envoie les informations : id =s22 ;β=2 ;γ=0 ;γm=0 ;f =∅ ;i=2 à la « machine 2 ».

Itération 5 — La « machine 1 » envoie les informations : id =s2 ;β=5 ;γ=0 ;γm=0 ;f ={s9, s10, s11} ;i=1 à la « machine 3 ».

— La « machine 1 » envoie les informations : id =s11 ;β=2 ;γ=0 ;γm=0 ;f =∅ ;i=1 à la « machine 2 ».

— La « machine 3 » envoie les informations : id =s11 ;β=3 ;γ=0 ;γm=0 ;f ={$s22$} ;i=2 à la « machine 2 ».

— La « machine 3 » envoie les informations : id =s15 ;β=3 ;γ=0 ;γm=1 ;f ={$s19, s11$} ;i=3 à la « machine 2 ».

Procedure 2 Reception de : (id,β, γ, γm, f, i) par la machine j

```
begin
1   foreach e ∈ S do
2       if (e.id == id) then
3           x.site[i].β ⟵ β
4           x.site[i].γ ⟵ γ
5           x.site[i].γm ⟵ γm
6           x.site[i].f ⟵ f
        end
    end
end
```

Les instructions décrivant la procédure de réception des valeurs, permettent de stocker les valeurs reçues sur l'état concerné. lorsque la machine reçoit les valeurs, l'état consterné est recherché (la ligne 1 présente l'instruction du parcours des états). Une fois l'état trouvé, les valeurs reçues sont stockées sur l'état dans l'ensemble « site », il contient les informations des machines sur cet état, l'indice de cet liste représente la clé de la machine.

Exemple 3.2.6. En appliquant le processus de réception les valeurs reçues sont stockées sur l'état concerné. Ainsi, nous remarquerons que les valeurs de certains états sont mises à jours avec les itérations. Les valeurs reçues pour chacun des états sont les suivantes :

Itération 1

— A la réception des valeurs : id =s9 ;β = 0 ;γ=1 ;γm=0 ;f ={S9} ;i=2 par la « machine 1 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S9 ».

— A la réception des valeurs : id =s10 ;β=2 ;γ=1 ;γm=1 ;f ={S10} ;i=2 par la « machine 1 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S10 ».

— A la réception des valeurs : id =s15 ;β=3 ;γ=0 ;γm=1 ;f ={$s19$} ;i=3 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S15 ».

Itération 2

— A la réception des valeurs : id =s9 ;β =6 ;γ =0 ;γm=0 ;f =∅, i=1 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S9 ».

— A la réception des valeurs : id =s10 ;β=5 ;γ=0 ;γm=0 ;f =∅ ;i=1 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S10 ».

— A la réception des valeurs : id =s7 ;β=2 ;γ=0 ;γm=0 ;f ={$s9$} ;i=1 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S7 ».

— A la réception des valeurs : id =s2 ;β=5 ;γ=0 ;γm=0 ;f ={$s9, s10$} ;i=1 par la « machine 3 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S2 ».

— A la réception des valeurs : id =s15 ;β=1 ;γ=0 ;γm=1 ;f =∅ ;i=2 par la « machine 3 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S15 ».

Itération 3 — A la réception des valeurs : id =s7 ;β=1 ;γ=0 ;γm=1 ;f =∅ ;i=1 par la « machine 1 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S7 ».

— A la réception des valeurs : id =s2 ;β=3 ;γ=0 ;γm=1 ;f =∅ ;i=3 par la « machine 1 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S2 ».

— A la réception des valeurs : id =s22 ;β=1 ;γ=0 ;γm=0 ;f ={$s2$} ;i=3 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S22 ».

Itération 4 — A la réception des valeurs : id =s11 ;β=2 ;γ=0 ;γm=0 ;f ={$s22$} ;i=2 par la « machine 1 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S11 ».

— A la réception des valeurs : id =s11 ;β=2 ;γ=0 ;γm=0 ;f ={$s22$} ;i=2 par la « machine 3 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S11 ».

— A la réception des valeurs : id =s22 ;β=2 ;γ=0 ;γm=0 ;f =∅ ;i=2 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S22 ».

Itération 5 — A la réception des valeurs : id =s2 ;β=5 ;γ=0 ;γm=0 ;f ={$s9$, $s10$, $s11$} ;i=1 par la « machine 3 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S2 ».

— A la réception des valeurs : id =s11 ;β=2 ;γ=0 ;γm=0 ;f =∅ ;i=1 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S11 ».

— A la réception des valeurs : id =s11 ;β=3 ;γ=0 ;γm=0 ;f ={$s22$} ;i=2 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S11 ».

— A la réception des valeurs : id =s15 ;β=3 ;γ=0 ;γm=1 ;f ={$s19$, $s11$} ;i=3 par la « machine 2 ». Ces dernières sont stockées à l'indice « i » de la variable « site » de l'état « S15 ».

3.2.7 Algorithme de Redistribution

Après l'envoi des valeurs calculées, les états *Border et Notifier* présentent les valeurs calculées aux paramètres β et γ par la machine locale et les machines distantes. Ces valeurs permettent d'appliquer une stratégie de redistribution des états. La stratégie adoptée est celle d'un jeux non coopératif. Chaque machines cherche a optimisée le temps de calcul de la vérification d'une formule en se focalisant sur l'information que pressente les valeurs calculées par chaque machine. Ainsi, grâce à ces valeurs, sur une machine est calculé calcule l'ensemble minimal des états à envoyer sur une machine distante ainsi que l'ensemble minimal des états à recevoir par cette machine locale. L'envoi des états nécessite la prise en compte de l'équilibrage de charge. Dans ce qui suit nous proposons une stratégie de recherche de ces ensembles minimaux d'états à déplacer, dans un sens ou dans l'autre, suivi d'une technique d'envoi de ces ensembles d'états.

L'algorithme de redistribution de l'espace d'états est décrit par l'algorithme 7.

Algorithm 7: Redistribution of States

1 Recherche de l'ensemble minimal des états à déplacer sur une machine distante;
2 Recherche de l'ensemble minimal des états à déplacer sur la machine locale;
3 Appliquée une heuristique de redistribution sur les deux ensembles;

Dans les sections suivantes, nous détaillons les phases de cet algorithme.

a Recherche de l'ensemble minimal des états à déplacer sur une machine distante

Dans cette partie l'algorithme cherche parmi les valeurs des paramètres β et γ calculées par les machines distantes le couple minimal. Le nombres des états associés à ce couple doit être inférieur au nombre des états associés au couple calculé par la machine locale. Le déplacement de ces états sur une machine distante permet de diminuer le temps de la vérification car l'algorithme pourra détecter la valeur logique de la formule sur ces états sans qu'une notification ne soit envoyée. Ainsi, le couple minimal est recherché sur les valeurs envoyées par les machines distantes au niveau des états Border *et* Notifier, comparées aux valeurs calculées par la machine locale sur cet état. La recherche du couple minimal est faite en premier temps sur les états ayant une dépendance mutuelle (c'est-à-dire les états distants dépendant l'un de l'autre) car le regroupement de ces états sur une machine diminue le nombre d'itérations de deux. Ainsi, lorsqu'il n'existe pas une dépendance mutuelle entre les états le couple minimum est recherché parmi les valeurs envoyées par les machines distantes au niveau des états Border *et* Notifier. La formalisation de ce principe est la suivante :

Function 2 $Search\_Min\_States\_DeplaceOnDistantMachine()$:state

```
begin
    MG ⟵ null
4   foreach e ∈ Border(F) do
        foreach e' ∈ Notifier(F) do
            for j = 1 to size(e'.site) do
                if ((e ∈ e'.f) and (exist(e.site[j])) and (e.β > e.site[j].β) and ((e'.β >
                e'.site[j].β) or (e'.β == e'.site[j].β and e'.γ > e'.site[j].γ)) and (((MG.β >
                e'.site[j].β) or (MG.β == e'.site[j].β and MG.γ > e'.site[j].γ) or MG ==
                null))) then
                    MG ⟵ e'.site[j]
    if MG == null then
5       foreach e ∈ {Border(F) ∪ Notifier(F)} do
            foreach site ∈ e.site do
                if ((e.β > site.β) or (e.β == site[j].β and e.γ > site.γ)) and (MG ==
                null or ((MG.β > site.β) or (MG.β == site[j].β and e.γ > site.γ))) then
                    MG ⟵ e'.site[j]
6   return MG
```

Les instructions décrivant la fonction précédente sont expliquées comme suit :

ligne ?? : L'instruction de cette ligne sert à initialiser la variable stockant les informations du couple minimal.

ligne 4 : Parcourir les états « *Border* » sur lesquels la formule n'es pas vérifiée. Ainsi, pour chaque état parcouru, les états « *Notifier* » sont parcourus à la ligne 4 pour rechercher le couple minimal parmis les valeurs envoyées par les machines distantes au niveau des états en dépendance mutuelle. Le parcours des valeurs envoyées par les machines distantes est fait à la ligne 4. Ainsi la vérification du couple minimal est faite à la ligne 4. Lorsqu'un couple minimal est trouvé le contenue de la variable « MG » est alors remplacé par les informations du nouveau couple (ligne 4).

ligne 4 : L'instruction de cette ligne vérifie qu'aucun minimum n'a été trouvé lors de la recherche précédente. Lorsque la condition est vérifiée les instructions de la ligne 5 jusqu'à la ligne 5 sont alors exécutées.

ligne 5 : L'instruction de cette ligne permet de parcourir les états « Border *et* Notifier » sur lesquels la formule n'es pas vérifiée.

ligne 5 : Pour chaque état, l'instruction de cette ligne parcoure les valeurs calculées par les machines distantes et recherche le couple minimum. La vérification du minimum est faite à la ligne 5. Ainsi la vérification de cette condition entraine l'exécution de l'instruction de la ligne 5, cela permet à remplacer l'ancien couple par le nouveau.

ligne 6 : L'instruction écrite sur cette ligne permet de retourner les informations du couple minimal.

b Recherche de l'ensemble minimal des états à déplacer

L'ensemble minimal des états à déplacer dans une machine locale correspond au couple minimum des couples(β et γ) calculés par la machine locale sur les états(Border *et* Notifier). Les états associés à ce couple entrainent un léger déséquilibre qui pourrait être accepté avec l'écart calculé. La recherche de ce couple (β et γ) est formalisée comme suit :

Function 3 $Search\_Min\_States\_DeplaceOnLocaleMachine(\mathrm{MG}:state):State$

```
begin
1   MF ⟵ null
2   foreach e ∈ {Border(S) ∪ Notifier(S)} − {MG} do
3       if (MG == null) then
4           foreach site ∈ x.site do
5               if ((x.β < site.β)or(x.β == site.β and (x.γ < site.γ))and(MF ==
                null or ((MF.β > x.β) or(MF.β == x.β and MF.γ > x.γ))) then
6                   MF ⟵ x
7       else
8           site ⟵ findById(MG.id, x.site)
9           if ((site ≠ null)and((MF == null)or(MF.β > x.β)or(MF.β == x.βandMF.γ >
                x.γ))and((x.β < site.β)or(x.β == site.βandx.γ < site.γ)) then
10              MF ⟵ x
11
12  return MF
```

Les instructions décrivant la fonction précédente sont expliquées comme suit :

ligne 1 : L'instruction de cette ligne sert à initialiser la variable stockant les informations du couple minimum.

ligne 2 : L'instruction de cette ligne permet de parcourir les états « Border *et* Notifier » sur lesquels la formule n'es pas vérifiée. L'état dont il est possible d'importer ses successeurs ou prédécesseurs à partir d'une machine distante est exclu du parcours.

ligne 3 : L'instruction de cette ligne vérifie qu'il n'existe pas des états à importer. La vérification de cette condition entraine l'exécution des instructions de la ligne 4 jusqu'à la ligne 8.

ligne 4 : L'instruction de cette ligne permet de parcourir les valeurs calculées par les machines distantes, ainsi les comparer avec les valeurs calculées par la machine locale, la comparaison est effectuée à la ligne 5. Lorsque les valeurs de la machine locale sont minimales l'instruction de la ligne 6 est exécutée, elle permet de mettre à jours le couple minimal.

ligne 7 : Lorsqu'il existe des états à déplacer sur une machine distante, un ensemble des états est alors recherché sur la machine locale pour l'envoyer sur la machine distante. La recherche de ce couple minimal est faite de la ligne 8 jusqu'à la ligne 11.

ligne 8 : L'instruction de cette ligne permet de récupérer sur l'état les valeurs envoyées par la machine distante. Les valeurs sont récupérées lorsque l'identité de la machine existe sur l'ensemble des machines de l'état.

ligne 9 : L'instruction de cette ligne vérifie l'existence des valeurs de la machine distante et leurs supérioritées à celle calculées par la machine locale sur cet état. Lorsqu'elles sont inférieurs au minimum courant, l'instruction de la ligne 10 met à jour le contenu de la variable « MF » par ce couple.

ligne 12 : L'instruction écrite sur cette ligne permet de retourner les informations du couple recherché.

c Heuristique de redistribution de l'espace d'états

Après la recherche de l'ensemble minimal des états à déplacer, l'ensemble d'états peut être déplacer lorsque l'équilibre peut être assurer entre les machines avec une faible duplication des états. Ainsi, le processus de vérification de l'équilibrage de charge est décris comme suit.

— Lorsque le résultat de la différence entre le nombre des états à déplacer sur machine locale et sur la machine distante est inférieur ou égale à l'écart calculé un message est alors envoyé pour demander les états a migré de la machine distante, comporte aussi les états provenant de cette machine locale.

— Sinon, si le nombre des états à déplacer est inférieur ou égale à l'écart alors la machine locale envoie ces états à déplacer à la machine distante.

— Sinon, s'il s'agit des états de la machine distante, la machine locale envoie une demande de ces états.

Ce principe est formalisé comme suit :

Procedure 3 $Send\_Request$(MG : $state$, MF : $state$)

```
begin
1   if | MG.site.β − MF.β |≤ δ then
2       send(element(MF), MG) to MG.id
3   else if MG.site.β ≤ δ then
4       send(null, MG) to MG.id
5   else if MF.β ≤ δ then
6       send(element(MF), null) to MF.id
```

Les instructions décrivant la procédure ($Send\_Request$) sont expliquées comme suit :

ligne 1 : L'instruction de cette ligne permet de vérifier qu'il est possible de déplacer des états locaux et états distants. La vérification de cette condition permet d'exécuter l'instruction de la ligne 2, elle permet d'envoyer les états à déplacer et la référence des états à importer.

ligne 3 : L'instruction de cette ligne permet de vérifier qu'il est possible de déplacer des états distants. La vérification de cette condition permet d'exécuter l'instruction de la ligne 4, elle permet d'envoyer la référence des états à déplacer à la machine distante.

ligne 1 : L'instruction de cette ligne permet de vérifier qu'il est possible de déplacer des états locaux. La vérification de cette condition permet d'exécuter l'instruction de la ligne 2, elle permet d'envoyer l'ensemble des états à la machine distante.

Function 4 $element(s : state)$

```
begin
1   states ← ∅
2   if type(s) == border then
3       foreach e' ∈ S do
4           if s.f ∈ e'.f and size(e.site) == 1 then
5               states ← states ∪ {e'}
7           else if s.f ∈ e'.limit then
8               states ← states ∪ {e'}
9               duplicate(e') in this machine
10  else if e ∈ notifier(S) then
11      states ← getSucc(s)  foreach e' ∈ e.f do
12          if type(e) ≠ border then
13              states ← states ∪ {duplicate(e)}
14          else
15              states ← states ∪ {e}
```

d Réception des états ou la demande des états à déplacer

A la réception d'une requête les états à insérer sont insérés et les ensemble des états demandés sont envoyés. Lorsque des états sont insérés à la structure de Kripke d'une machine, les valeurs calculées par la machine distante sont alors mises à jour par celles correspondantes à la machine locale pour le bon fonctionnement du protocole. Après ce processus l'Algorithme 7 est relancé. Ce principe est formalisé comme suit :

Procedure 4 $Receive\_Request(states : Listofstate, \text{MG} : state)$

```
begin
1   | if size(states) ≠ 0 then
2   |   | add(states) in Sj
    | end
4   | if MG ≠ null) then
5   |   | send(element(MG), null) to MG.id
    | end
7   | algorithm 7()
end
```

Les instructions décrivant la procédure ($Receive\_Request$) sont expliquées comme suit :

ligne 1 : L'instruction de cette ligne vérifie que des états ont été envoyés. La vérification de cette condition permet d'exécuter l'instruction de la ligne 2, elle permet de rajouter les états reçus à la structure de Kripke(ligne 2).

ligne 3 : L'instruction de cette ligne vérifie qu'une demande concernant un ensemble des états a été envoyé. La vérification de cette condition permet d'envoyer l'ensemble des états concernant la demande (ligne 4).

ligne 5 : L'instruction sur cette ligne permet de relancer le protocole de redistribution afin de rechercher un ensemble des états à déplacer.

Exemple 3.2.7. L'exécution du protocole de redistribution sur les résultats générés dans les exemples précédents permet d'obtenir les résultats suivants :

Itération 1 : Pendant la première itération du protocole sur la « machine 1 », l'algorithme détecte que les état « S7 » et « S9 » sont mutuellement dépendants. Ainsi, les valeurs envoyées par la « machine 2 » sur l'état « S7 » sont minimales, les états concernant ces valeurs sont à déplacer de la « machine 2 ». Ainsi les états à déplacer de la « machine 1 » vers la « machine 2 » sont recherchés, l'algorithme détecte que les états liés à l'état « S11 » sont minimaux. Les états de cet ensemble sont à envoyer sur la « machine 2 ». Après ces deux processus de recherche, les valeurs sont évaluées afin de ressortir la requête à envoyer. L'évaluation montre qu'il est possible d'importer les prédécesseurs de l'état « S7 » et les prédécesseurs de l'état « S11 ». L'état « s2 » ne sera pas envoyé parmi les états à déplacer car la « machine 3 » possède des prédécesseurs directs de cet état. Après l'envoi des états à déplacer, la « machine 1 » possède l'état « S9 », quant à l'état « S1 » il est possédé par la « machine 2 ».

Sur la « machine 3 » l'algorithme détecte qu'il est possible de déplacer l'état « S22 » sur la « machine 2 ». Après ces redistributions d'états, le protocole détecte qu'il est impossible de redistribuer les états car il risque d'avoir un déséquilibre de charge sur les machines.

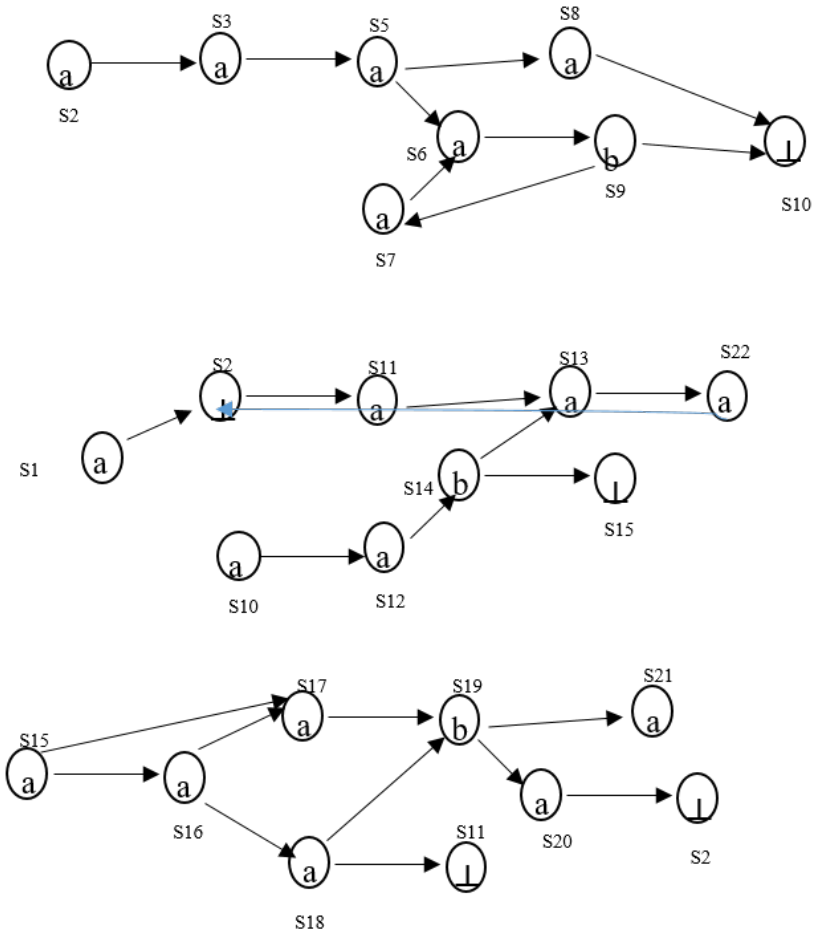


FIGURE 3.4 – Structure de kripke Redistribué

3.3 Étude expérimentale

Cette section présente les différents outils utilisés à l'aboutissement de l'application réalisée ainsi que l'architecture de l'application développée. A partir de l'application développée nous réalisons une étude expérimentale d'un exemple issu de la vie courante, ensuite nous interprétons les résultats expérimentale obtenus par la méthode de redistribution proposée dans ce chapitre.

3.3.1 Outils de développement

Pour la mise en œuvre d'une application de distribution de l'espace d'états et de la vérification formelle, la capacité à monter en charge s'envisage pour pouvoir évoluer de façon fluide en fonction de l'augmentation de la demande des ressources afin qu'une expérimentation soit menée dans les meilleures conditions. Dans ce qui suit nous pressentons les outils nécessaires pour le développement de ce type d'application.

Langage de programmation Java

Java est un langage de programmation moderne orientée objet développé par Sun Microsystems (aujourd'hui racheté par Oracle) [**?**]. Le langage Java permet la programmation parallèle, le développement des applications reparties, l'accès à des bases de données, l'accès à des traitements en d'autres langages. Ainsi, les applications développées sous le langage Java sont portables sur plusieurs systèmes d'exploitation tels que Unix, Windows, Mac OS ou GNU/Linux. Par contre elles sont plus lourdes à l'exécution (en mémoire et en temps processeur) à cause de sa machine virtuelle.

La plateforme JADE

JADE (Java Agent Development Framework) est une plateforme de programmation multi-agent implémentée en Java [**?**]. Elle est open-source et distribuée par Telecom Italia sous la licence LGPL. La plateforme héberge un ensemble d'agents, identifiés de manière unique, pouvant communiquer de manière bidirectionnelle avec les autres agents. Ainsi Chaque agent s'exécute dans un conteneur (container) qui lui fournit son environnement d'exécution.

IDE IntelliJ

IntelliJ est un environnement de développement intégré (IDE) pour le développement de logiciels. Il propose de nombreux outils pour faciliter le développement dont l'auto-complétion syntaxique, une vérification d'erreur en live, différents outils de compilation, des outils de debugging avancés, etc. IDE IntelliJ est adapté pour coder en Java, en plus de java plusieurs plugins sont offert pour le développement logiciels.

Docker

Docker une plateforme logicielle open source permettant la mise en œuvre de systèmes distribués. Il permet à de multiples applications, tâches de fond et autres processus de s'exécuter de façon autonome sur une seule machine physique ou à travers un éventail de machines isolées. Il a été développé par Solomon Hykes avec les contributions d'Andrea Luzzardi et Francois-Xavier Bourlet [**?**]. Les services ou fonctions de l'application et ses différentes bibliothèques, fichiers de configuration, dépendances et autres composants sont regroupés au sein d'un container. Chaque container exécuté partage les services du système d'exploitation de la machine physique.

Neo4j

Neo4j est une base de données graphe [**?**], sous licence GPLv3, implémenté en Java et un langage de requête de graphe déclaratif simple et efficace (Cypher). Développée par Neo Technology dans le but d'offrir une structure de stockage dédiées aux graphes ainsi qu'une parcours des données en utilisant les arcs pour passer d'un nœud à l'autre. De plus, il offre une haute disponibilité des données et le stockage des milliards de nœuds et de relations.

GraphViz

GraphViz (Graph Visualiszation Software) est un ensemble d'outils de visualisation de graphes, créés par les laboratoires de recherche d'AT&T [**?**]. Il utilise des fichiers textes suivant le langage DOT pour représenter les données structurées sous forme de graphes. le langage DOT permet de personnaliser le rendu des graphes par le choix des formes, couleurs et polices de caractères. Ainsi le graphe définie est exporté sous différents formats d'images (FIG, PNG, JPEG, GIF, etc.).

3.3.2 Mise en œuvre

Nous avons développé une application distribuée implémenter avec le langage de programmation JAVA sur l'IDE IntelliJ. Chaque instance de l'application dispose une base de donnée orienté graphe (Neo4j) pour le stockage de l'espace d'états. Les applications de même réseau sont utilisés pour accomplir une tache de vérification ou de génération de l'espace d'états.

L'application développée pour l'expérimentation de notre approche dispose d'un éditeur graphique pour modéliser le réseau de Pétri du système à analyser (Figure 3.7). L'outil permet de générer la structure de Kripke distribuée d'un réseau de Pétri modélisé. Il permet d'exécuter le model checking sur une structure de Kripke distribuée ainsi que la redistribution de l'espace d'états par l'approche proposée (Figure **??**).

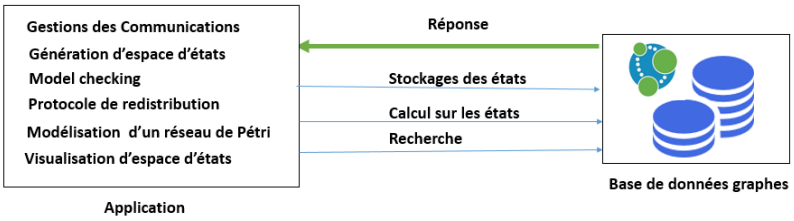


FIGURE 3.5 – Architecture de l'application développée

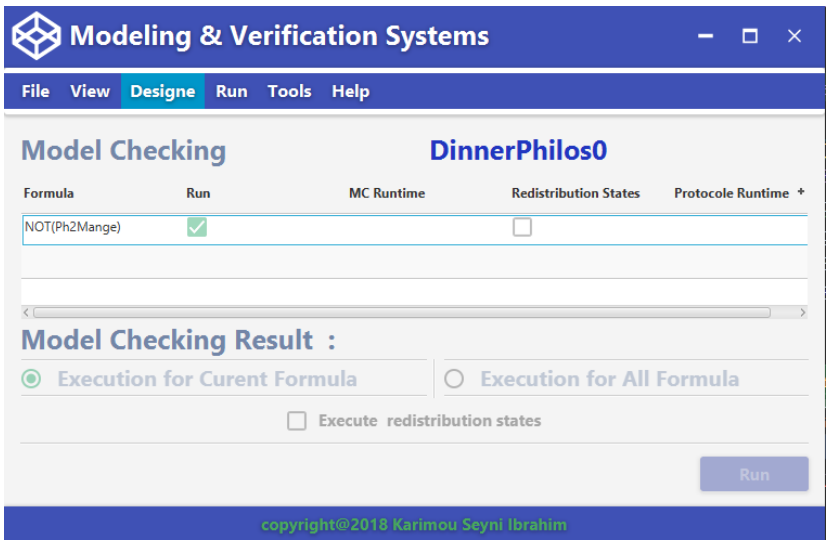


FIGURE 3.6 – Interface d'exécution model checking

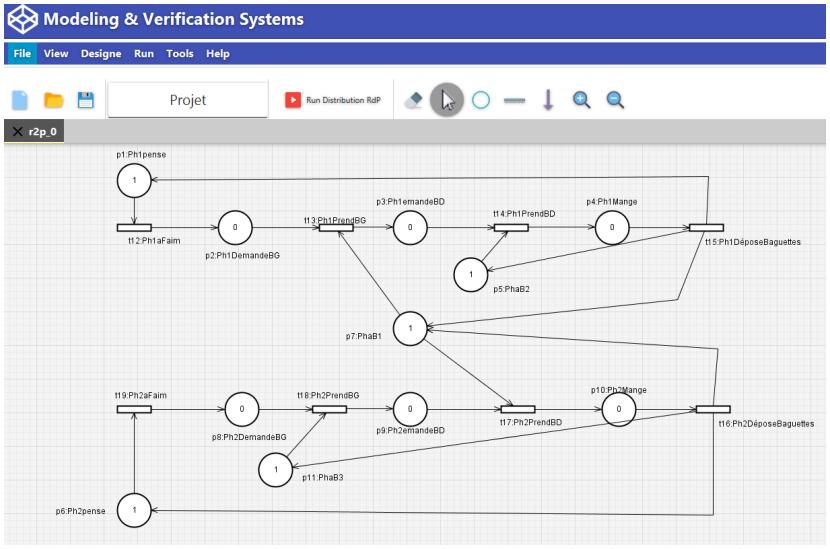


FIGURE 3.7 – Éditeur de modélisation d'un réseau de Pétri

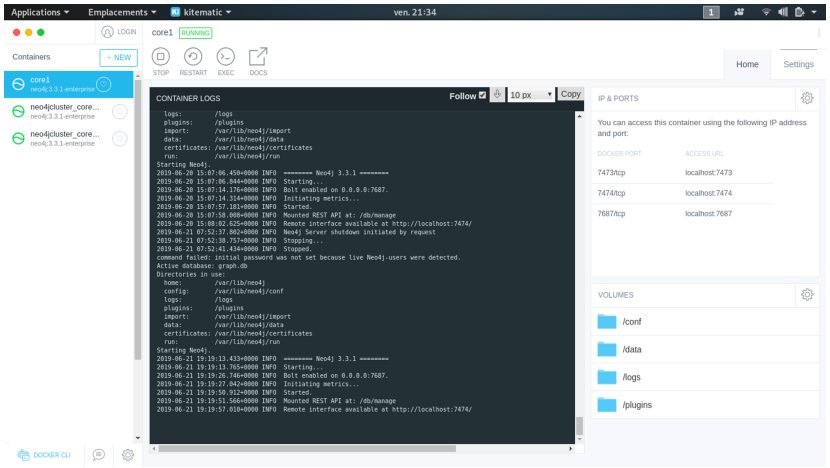


FIGURE 3.8 – Virtualisation des Bases de données Neo4j sur Doker

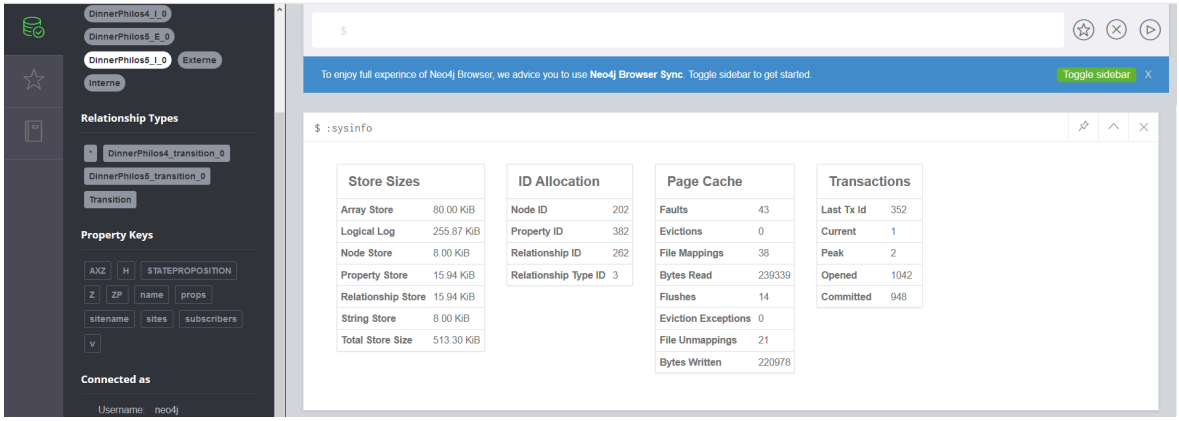


FIGURE 3.9 – Base de données Neo4j

3.3.3 Expérimentation

Nous avons évalué l'efficacité de la politique de redistribution proposée avec l'approche de distribution basée sur la fonction de hachage MD5 sur le modèle des philosophes. A partir des tableaux 3.17 et 3.18, on constate que l'approche proposée assure une bonne distribution de l'espace d'états pour le model checking tout en maintenant un bon équilibrage de charge, ce qui signifie que toutes les machines du réseau contiennent le nombre d'états qu'il peuvent supporter. En plus du nombre d'états on constate que le temps de la vérification s'améliore de plus en plus, ce qui indique qu'aucune machine n'est surchargée en mémoire et en calcul.

Philos	Formules	Machines	Temps d'exécution	Etats Interne	Etats Externe
2 Philos	NOT(Ph2Mange)	Machine 1	2 s	2	2
		Machine 2	2 s		
	NOT(Ph1Mange)	Machine 1	2 s	6	2
		Machine 2	10 s		
3 Philos	NOT(Ph2Mange)	Machine 1	2 s	17	14
		Machine 2	2 s		
	NOT(Ph1Mange)	Machine 1	19 s	15	14
		Machine 2	10 s		
4 Philos	NOT(Ph2Mange)	Machine 1	10 s	35	53
		Machine 2	19 s		
		Machine 3	32 s	40	49
	NOT(Ph1Mange)	Machine 1	35 s		
		Machine 2	10 s	37	46
		Machine 3	19 s		

TABLEAU 3.17 – Résultats du model checking sur l'espace d'états distribués avec **MD5**

Philos	Formules	Machines	Exécution	Etats Interne	Etats Externe	Duplicata
2 Philos	NOT(Ph2Mange)	Machine 1	2 s	2	2	1
		Machine 2	2 s			
	NOT(Ph1Mange)	Machine 1	2 s	6	2	0
		Machine 2	2 s			
3 Philos	NOT(Ph2Mange)	Machine 1	2 s	17	14	14
		Machine 2	2 s			
	NOT(Ph1Mange)	Machine 1	2 s	15	14	10
		Machine 2	2 s			
4 Philos	NOT(Ph2Mange)	Machine 1	2 s	35	53	39
		Machine 2	2 s			
		Machine 3	2 s	40	49	36
	NOT(Ph1Mange)	Machine 1	2 s			
		Machine 2	2 s	37	46	33
		Machine 3	2 s			

TABLEAU 3.18 – Résultats du model checking sur l'espace d'états redistribués avec **CDS**

3.4 Discussion

Sur la base des résultats expérimentaux présentés ci-dessus, nous constatons l'efficacité de l'approche proposée pour la distribution de l'espace d'états. En termes d'équilibrage l'approche proposée s'est avérée efficace à cause de sa capacité d'analyser le comportement du système et à extraire les informations pertinentes sur ses états. Ainsi, les résultats ont démontrés que la minimisation des transitions externe n'implique pas toujours la minimisation du temps de calcul. Cependant, Notre approche n'est pas dédiée à réduire le nombre de transitions externes, mais d'analyser les états afin de proposer une distribution adéquat des états tout en garantissant un nombre minimal de communication et un équilibrage de charge entre les machines.

3.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche pour la distribution de l'espace d'états basée sur le comportement du système analysé. L'approche proposée vise à améliorer la distribution de l'espace d'états qui sera bénéfique pour le model checking car entraine moins de communication entre les machines. L'approche proposée analyse le comportement d'un système donné, et extrait les informations pertinentes sur ses états. Ensuite, les états sont redistribués suite à leurs pertinences soit migré définitivement soit dupliqués sur d'autres machines, afin de minimiser le nombre de communications entre les machines. La machine récepteur de ces états est choisi suite à la une stratégie du jeux non coopérative où chaque machine cherche à minimiser le taux de ses communications tout en maintenant un bon équilibrage des états entre les machines à l'aide des seuils prédéfinis pour chaque machine.

L'approche proposée peut être adoptée à tout autre spécification formelle ou modèles d'analyse de données car il pratiquement possible de la formalisée comme un graphe.

Conclusion et Perspectives

Conclusion

La vérification formelle constitue une étape indispensable pour garantir le bon fonctionnement des systèmes complexes et critiques. Ainsi, l'approche basée sur le model checking permet de vérifier des propriétées sur un système décrit avec un modèle formel. Cette approche souffre d'un problème majeur engendré par l'explosion combinatoire de l'espace d'états à explorer dans un temps raisonnable pour certains systèmes spécifiés.

La distribution est une solution pour combattre l'explosion combinatoire de l'espace d'états. Cette distribution consiste à construire cet espace d'états sur un réseau de machines connectées. Les travaux traitant cette distribution se sont focalisés sur un aspect précis du problème tel que l'équilibrage de charge, la minimisation des transitions lient les différentes parties. En plus de ces travaux une première approche de redistribution comportementale a été proposé basée sur la pertinence des états et leurs connexions (transitions internes et externes).

Dans ce travail, nous avons proposé une nouvelle approche de redistribution de l'espace d'états pour le model checking distribué basé sur la théorie de jeux et la pertinence des états. La compréhension du fonctionnement du model checking distribué a permis de constater que la diminution de transitions liant les différentes parties peut entraîner une mauvaise qualité de distribution de l'espace d'états car la validitée d'une formule sur un état dépendra de ces états successeurs.

Partant de ce constat, nous proposons une nouvelle politique de redistribution qui entre dans le cadre de la théorie de jeux. L'approche propose vise a analysée le comportement des systèmes et redistribuer les états pertinents selon une politique basé sur les jeux non coopérative pour optimiser les performances du système. Chaque machine cherche a optimisée ces performances en définissant une bonne localité pour un état pertinent tout en optimisant l'équilibrage de charge et la quantité de communication entre les machines. Les expérimentations de cette approche ont montré son efficacité de se rapprocher de la meilleure distribution de l'espace d'états au cours des exécutions future, ce qui accélère d'avantage le processus de vérification des propriétés tout en garantissant l'équilibrage de charge et un taux de communication minimale. Suite aux résultats l'approche, il est possible que les machines sont utilisées au maximum de leur capacité et donc on peut arriver à une accélération linéaire.

Perspectives

Le travail présenté dans ce projet de fin d'étude a permis de dégager plusieurs perspectives à développer dans l'avenir :

— Un premier objectif consiste à explorer les différentes stratégies de la théorie de jeux en apportant des améliorations supplémentaires à notre stratégie afin d'avoir une meilleur stratégie qui converge directement à une meilleur distribution.

— Nous pensons qu'il peut être intéressant d'utiliser les différentes statistiques générer par l'exécution du model cheking pour extraire un modèle de partitionnement base sur le machine learning.

Références Bibliographies

ABIDINE, B. Z. 2011, *Split of Territories in Concurrent Optimization*, thèse de doctorat, Contantine 1. 26, 29, 37

ALLMAIER, K. M. E. H. G., S. C. 1997, «State space construction and steady-state solution of gspns on a shared-memory multiprocessor», *In Petri Nets and Performance Models, 1997., Proceedings of the Seventh International Workshop on*. 18

AREIBI, Z., S. ET YANG. 2004, «Effective memetic algorithms for design= genetic algorithms+ local search+ multi-level clustering», *Evolutionary Computation*. 15

ARMBRUSTER, F. M. H. C. E. M., M. 2008, «A comparative study of linear and semidefinite branch-and-cut methods for solving the minimum graph bisection problem», *In IPCO*. 13

AURORA, P. K. 2007, *Multi-level graph partitioning*, thèse de doctorat, Thèse de doctorat,University of Florida. 16

BARAT, C. C. E. P. F., R. 2016, «Partitionnement multi-critères de graphes pour l'équilibrage de charge de simulations multi-physiques», *In Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*. 13

BARNARD, H. D., S. T. ET SIMON. 1994, «Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems», *Concurrency and computation : Practice and Experience*. 14, 15

BATTITI, A. A., R. ET BERTOSSI. 1999, «Greedy, prohibition, and reactive heuristics for graph partitioning», *IEEE Transactions on Computers*. 15

BENSETIRA, I. 2017, *Proposition d'algorithmes de distribution des espaces d'états en vue d'une vérification basée model checking : Application aux automates temporisés avec durées d'actions*, thèse de doctorat, Université Abdelhamid Mehri - Constantine 2. ii, 2, 14, 17, 18, 19, 34

BICHOT, A. J.-M. D. N. E. B. P., C.-E. 2004, «Optimisation par fusion et fission, application au probleme du découpage aérien européen», *Journal Européen des Systèmes Automatisés (JESA)*. 15

BICHOT, C.-E. 2007, *Élaboration d'une nouvelle méta heuristique pour le partitionnement de graphe : la méthode de fusion-fission. Application au découpage de l'espace aérien*, thèse de doctorat, TOULOUSE. ii, 10

BICHOT, C. E. S. 2013, «Graph partitioning», . 13

BLOM, S., S. ET ORZAN. 2005, «A distributed algorithm for strong bisimulation reduction of state spaces», *International Journal on Software Tools for Technology Transfer (STTT)*. 18

BONAMI, N. V. H.-K. M. E. M. M., P. 2012, «On the solution of a graph partitioning problem under capacity constraints», *ISCO*. 13

BOPPANA, R. B. 1987, «Eigenvalues and graph bisection : An average-case analysis», *In Foundations of Computer Science, 1987., 28th Annual Symposium on*. 14

BOUZENADA, B. M. G.-N. R. A. E. S. D. E., M. 2012, «A generalized graph strict strong coloring algorithm», *International Journal of Applied Metaheuristic Computing*. 18

BUI, B. R., T. N. ET MOON. 1996, «Genetic algorithm and graph partitioning», *IEEE Transactions on Computers*. 15

BULUÇ, M. H. S. I. S. P. E. S. C., A. 2016, «Recent advances in graph partitioning», *In Algorithm Engineering*. 13

CHAN, M. Y. L. Y. Y. T. P. E. T. R. K., J. J. M. 2016, «Parallel ant brood graph partitioning in julia», *In Parallel Processing and Applied Mathematics*. 16

CHEVALIER, F., C. ET PELLEGRINI. 2006, «Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework», *Euro-Par 2006 Parallel Processing*. 15

CHEVALIER, F., C. ET PELLEGRINI. 2008, «Pt-scotch : A tool for efficient parallel graph ordering», *Parallel computing*. 16

CHEVALIER, I., C. ET SAFRO. 2009, «Comparison of coarsening schemes for multilevel graph partitioning», *Learning and Intelligent Optimization*. 16

CHOCKALINGAM, S., T. ET ARUNKUMAR. 1992, «A randomized heuristics for the mapping problem : The genetic approach», *Parallel computing*. 15

CIARDO, G. J. E. N. D., G. 1998, «Distributed state space generation of discrete-state stochastic models», *INFORMS Journal on Computing*. 18

CONG, R. M. E. X. M., J. 2003, «Optimality, scalability and stability study of partitioning and placement algorithms», *In Proceedings of the 2003 international symposium on Physical design*. 12

DELLING, G. A. V. R. I. E. W. R. F., D. 2012, «Exact combinatorial branch-and-bound for graph bisection», *In 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. 13

DEO, N. 2017, *Graph theory with applications to engineering and computer science*, Courier Dover Publications. 7

DESIDER, J.-A. 2007, «Split of territories in concurrent optimization», *inria*, vol. 32, n° 1, p. 286–295. 39

DHILLON, G. Y. E. K. B., I. S. 2007, «Weighted graph cuts without eigenvectors a multilevel approach», *IEEE transactions on pattern analysis and machine intelligence*. 13

DONATH, A., W. ET HOFFMAN. 1972, «Algorithms for partitioning of graphs and computer logic based on eigenvectors of connections matrices», *IBM Technical Disclosure Bulletin*. 14

E. A. EMERSON, J. Y. H. 1983, «'sometimes' and hot never' revesited on branching versus linear time», *11th Symposium on Principles Of Programnling Languages, Austin,Texas*. 27

E. M. CLARKE, E. A. E. 1981, «Design and synthesis of synchronizatioll skdctons using branching time temporal logic», *Proc. of an IBM Workshop on Logic of Programs, LNCS.* 27

EDMUND M. CLARKE, O. G. E. D. A. P. 1999, «Model checking», *Cambridge, MIT Press.* 25

EULER. 1736, «"solutio problematis ad geometriam situs pertinentis". commentarii academiae scientarum imperialis petropolitanae», •. 7

FELDMANN, A. E. E. W. 2015, «An o(n^4) time algorithm to compute the bisection width of solid grid graphs», *Algorithmica.* 13

FERREIRA, M. A. D. S. C. C. W. R. E. W. L. A., C. E. 1998, «The node capacitated graph partitioning problem : a computational study», *Mathematical Programming.* 13

FIDUCCIA, R. M., C. M. ET MATTHEYSES. 1988, «A linear-time heuristic for improving network partitions», *In Papers on Twenty-five years of electronic design automation.* 14

FIEDLER, M. 1973, «Algebraic connectivity of graphs», *Czechoslovak mathematical journal. Institute of Mathematics, Academy of Sciences of the Czech Republic.* 14

FIEDLER, M. 1975, «A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory», *Czechoslovak mathematical journal. Institute of Mathematics, Academy of Sciences of the Czech Republic.* 14

GARAVEL, M. R. E. S. W., H. 2013, «Large-scale distributed verification using cadp : Beyond clusters to grids», *Electronic Notes in Theoretical Computer Science.* 18, 21

GAREY. 1976, «Some simplified np-complete graph problems», *Theoretical computer science.* 12, 13

GRADY, E. L., L. ET SCHWARTZ. 2006, «Isoperimetric graph partitioning for image segmentation.», *IEEE transactions on pattern analysis and machine intelligence.* 13

GUIDOUM, B. M. E. S. D. E., N. 2013, «The strict strong coloring based graph distribution algorithm», *International Journal of Applied Metaheuristic Computing.* 18

HAGEN, H. D.-H. E. K. A. B., L. W. 1997, «On implementation choices for iterative improvement partitioning algorithms», *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* 14

HAGER, P. D. T. E. Z. H., W. W. 2013, «An exact algorithm for graph partitioning», *Mathematical Programming.* 13

HENDRICKSON, R., B. ET LELAND. 1995a, «An improved spectral graph partitioning algorithm for mapping parallel computations», *SIAM Journal on Scientific Computing.* 14

HENDRICKSON, R., B. ET LELAND. 1995b, «A multi-level algorithm for partitioning graphs», *Supercomputing.* 15

HOLYER, I. 1981), «The np-completeness of some edge-partition problems», *SIAM Journal on Computing.* 13

HYAFIL, R. L., L. ET RIVEST. 1973, «Graph partitioning and constructing optimal decision trees are polynomial complete problems», *Technical Report. IRIA. Laboratoire de Recherche en Informatique et Automatique.* 13

J. W. BAKKER, J. J. C. K., J. A. BERGSTRA. 1983, «Linear the and branching time semantics for recursion with merg», *Proc. ICALP 83, LNCS.* 27

JOHNSON, A. C. R. M. L. A. E. S. C., D. S. 1989, «Optimization by simulated annealing : an experimental evaluation ; parti, graph partitioning», *Operations research.* 15

JOVANOVIC, T. M. E. V. S., R. 2016, «An ant colony optimization algorithm for partitioning graphs with supply and demand», *Applied Soft Computings.* 15

KABELÍKOVÁ, P. 2006, *Graph partitioning using spectral methods*, thèse de doctorat, VSB - Technical University of Ostrava, Czech Republic. 14

KAIBEL, P. M. E. P. M. E., V. 2011, «Orbitopal fixing», *Discrete Optimization.* 13

KALAYCI, R., T. E. ET BATTITI. 2018, «A reactive self-tuning scheme for multilevel graph partitioning», *Applied Mathematics and Computation.* 16

KARYPIS, G. 2003, «Multilevel hypergraph partitioning», *In Multilevel Optimization in VLSICAD.* 16

KARYPIS, G. 2011, «Metis and parmetis», *In Encyclopedia of parallel computing.* 16

KARYPIS, V., G. ET KUMAR. 1998, «A parallel algorithm for multilevel graph partitioning and sparse matrix ordering», *Journal of Parallel and Distributed Computing.* 16

KERNIGHAN, S., B. W. ET LIN. 1970, «An efficient heuristic procedure for partitioning graphs», *The Bell system technical journal. Alcatel-Lucent.* 14

KIM, H. I. K. Y.-H. E. M. B.-R., J. 2011, «Genetic approaches for graph partitioning : a survey», *In Proceedings of the 13th annual conference on Genetic and evolutionary computation.* 15

KOROŠEC, I. J. E. R. C. B., P. 2004, «Solving the mesh-partitioning problem with an ant-colony algorithm», *Parallel computing.* 15

KRIPKE, S. A. 1963, «Semantical considerations on modal logic», *Acta Philosophica Fennica Fasc.* 25

KRISHNAMURTHY, B. 1984, «An improved min-cut algonthm for partitioning vlsi networks», *IEEE Transactions on Computers.* 14

L, N. 1989, *Discrete Mathematics*, Oxford University Press. 8

LAND, A. G., A. H. ET DOIG. 2010, «An automatic method for solving discrete programming problems», *50 Years of Integer Programming 1958-2008.* 14

LANGHAM, P., A. E. ET GRANT. 1999, «A parallel genetic algorithm for the graph partitioning problem», *In Proceedings of the 5th international conference on Supercomputing.* 15

LASALLE, G., D. ET KARYPIS. 2013, «Multi-threaded graph partitioning», *In Parallel and Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* 16

LASALLE, G., D. ET KARYPIS. 2015, «Multi-threaded modularity based graph clustering using the multilevel paradigm», *Journal of Parallel and Distributed Computing*. 16

LENG, S., M. ET YU. 2007, «An effective multi-level algorithm based on ant colony optimization for bisecting graph», *In Pacific-Asia Conference on Knowledge Discovery and Data Mining*. 16

MEYERHENKE, S., H. ET SCHAMBERGER. 2006, «A parallel shape optimizing load balancer», *Euro-Par 2006 Parallel Processing*. 15

MONIEN, P. R. E. D. R., B. 2000, «Quality matching and local improvement for multilevel graph-partitioning», *Parallel Computing*. 16

NASH, J. 1950, «The bargaining problem», *Econometrica*, vol. 18, n° 1, p. 155–162. 39

PELLEGRINI, F. 1995, *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*, thèse de doctorat, Bordeaux 1. 16

PELLEGRINI, F. 2007, «A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries», *Euro-Par 2007 Parallel Processing*. 15

PETRI, C. A. 1962, *Communication avec des Automates*, thèse de doctorat, Allemagne à Bonn en. 21

POPE, T. D. R. E. K. A. D., A. S. 2000, «Evolving multi-level graph partitioning algorithms», *In Computational Intelligence (SSCI)*. 16

POTHEN, S. H. D. E. L. K.-P., A. 1990, «Partitioning sparse matrices with eigenvectors of graphs», *SIAM journal on matrix analysis and applications*. 14

PREDARI, E. A. E. R. J., M. 2017, «Comparison of initial partitioning methods for multilevel direct k-way graph partitioning with fixed vertices», *Parallel Computing*. 16

QUEYROI, F. 2013, *Partitionnement de grands graphes : mesures, algorithmes et visualisation*, thèse de doctorat, Université Sciences et Technologies - Bordeaux I. 10

RAHIMIAN, P. A. H. G. S. J.-M. H.-S., F. 2013, «Ja-be-ja : A distributed algorithm for balanced graph partitioning», *In Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*. 16

RODRIGUES, B. P. E. C. J. M. D. F. J. C. E. G. D. D., C. L. 2006, «A bag-of-tasks approach for state space exploration using computational grids», *In Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*. 18

SAFRO, S. P. E. S. C., I. 2015, «Advanced coarsening schemes for graph partitioning», *Journal of Experimental Algorithmics*. 16

SAIDOUNI, C. A. C. E. I. J.-M., D. E. 2012, «On the fly pso inspired algorithm for graph distribution», *In 2nd International Symposium on Modelling and Implementation of Complex Systems*. 18, 25, 34

SANDERS, P. E. S. 2011, «Engineering multilevel graph partitioning algorithms», *In ESA*. ii, 16

SANDERS, P. E. S. 2012, «Distributed evolutionary graph partitioning», *In 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. 16

SCHLOEGEL, K. G. E. K., K. 2000, «Graph partitioning for high performance scientific simulations», *Army High Performance Computing Research Center*. 13

SCHNEIDER, K. 2004, «Verification of reactive systems : Formal methods and algorithms», *Springer-Verlag, coll. « Texts in Theoretical Computer Science. An EATCS Series »*. 26

SELLMANN, S. N. E. T. L., M. 2003, «Multicommodity flow approximation used for exact graph partitioning», *In ESA*. 13

SIMON, H. D. 1991, «Partitioning of unstructured problems for parallel processing», *Computing systems in engineering*. 14

SOPER, W. C. E. C. M., A. J. 2004, «A combined evolutionary search and multilevel optimisation approach to graph-partitioning», *Journal of Global Optimization*. 15

SØRENSEN, M. M. 2003, «Facet-defining inequalities for the simple graph partitioning polytope», *Discrete Optimization*. 13

T, M. 1989, «Petri nets : Properties, analysis and applications», *Proceedings of the IEEE*. 23

TABIB, D. E., N. ET SAIDOUNI. 2016, «Newton's law of universal gravitation based genetic algorithm for graph distribution», *International Journal of Computational Intelligence and Applications. World Scientific*. 19, 34

TABIB, N. 2017, *Construction distribuée d'espace d'états en vue d'une vérification efficace des systèmes atemporels*, thèse de doctorat, Thèse de doctorat, Université Abdelhamid Mehri - Constantine 2, Algeria. 19

TALBI, P., E.-G. ET BESSIERE. 1991, «A parallel genetic algorithm for the graph partitioning problem», *In Proceedings of the 5th international conference on Supercomputing*. 15

TALU, M. F. 2017, «Multi-level spectral graph partitioning method», *Journal of Statistical Mechanics : Theory and Experiment*. 16

TAO, Y., L. ET ZHAO. 1993, «Multi-way graph partition by stochastic probe», *Computers and operations research*. 15

TASHKOVA, K. P. E. I. J., K. 2011, «A distributed multilevel ant-colony algorithm for the multi-way graph partitioning», *International Journal of Bio-Inspired Computation*. 16

TRIFUNOVIC, W. J., A. ET KNOTTENBELT. 2015, «Parkway 2.0 : A parallel multilevel hypergraph partitioning tool», *Lecture notes in computer science*. 16

VALETTE, R. 9novembre 2007, «Genèse de la théorie des réseaux de petri», *LAAS-CNRS Toulouse*. 22

VAN DRIESSCHE, D., R. ET ROOSE. 1994, *A graph contraction algorithm for the calculation of eigenvectors of the laplacian matrix of agraph with a multilevel method*, thèse de doctorat, Rapport technique TW 209, Department of Computer Science, KU Leuven, Leuven, Belgium. 15

WALSHAW, C. «Jostle-graph partitioning software», URL http://staffweb.cms.gre.ac.uk/wc06/jostle/. 16

WAN, R. S. S. A. E. L. B., Y. 2005, «A stochastic automaton-based algorithm for flexible and distributed network partitioning», *In Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE.* 15

WILLIAMS, R. D. 1991, «Performance of dynamic load balancing algorithms for unstructured mesh calculations», *Concurrency and Computation : Practice and Experience. Wiley Online Library.* 15