Num. d'ordre : 07/2012-E/INF

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Electronique et d'Informatique

Thèse de Doctorat d'état

Informatique

Présentée par
Mohand Cherif BOUKALA

Thème

Vérification distribuée des systèmes complexes

Soutenue publiquement le 08 juillet 2012, devant le Jury :

Prof. N. Badache	USTHB	Président
Prof. A. Mokhtari	USTHB	Directrice de Thèse
Prof. L. Petrucci	LIPN Paris 13	Co-Directrice
Prof. M. Ahmed Nacer	USTHB	Examinateur
Prof. L. Sekhri	Univ. Oran	Examinateur
Prof. A. Chaoui	Univ. de Constantine	Examinateur

Remerciements

Je tiens tout d'abord à remercier le Professeur Nadjib BADACHE pour l'honneur qu'il me fait en acceptant de présider le jury de cette thèse. je lui suis trés reconnaissant pour l'interêt qu'il a toujours accordé à mes travaux.

Je remercie le Professeur Mohamed AHMED NACER, Directeur du Laboratoire des Systèmes Informatiques (LSI) de s'être intéressé à mes recherches et d'avoir accepté d'être dans ce jury.

Je remercie vivement les Professeurs Larbi SEKHRI et Allaoua CHAOUI de consacrer de leur temps pour la lecture de mon travail. Je leur suis vivement reconnaissant d'avoir acceptè de participer au jury.

Je remercie le Professeur Aicha AISSANI pour avoir dirigé cette thèse, et pour sa contribution active dans l'accomplissement de ce travail. Son expérience, ses critiques et ses encouragements ont été très bénéfiques.

Je tiens à exprimer toute ma gratitude et ma reconnaissance au Professeur Laure PETRUCCI qui a encadré mes recherches. Ses travaux personnels ont servi de point de départ à cette thèse. Nos discussions, toujours très fructueuses, ses suggestions pertinantes ont permis de compléter ma recherche de manière efficace. Les séjours que j'ai passé au sein de son laboratoire, au LIPN, ont beaucoup compté dans mes recherches et dans l'aboutissement de cette thèse.

Je rend un grand hommage au Professeur Daniel Keyser pour avoir initié une coopération entre le LIPN avec notre département, grâce à laquelle j'ai pu mener à terme cette thèse. Je n'oublie pas l'accueil chaleureux que j'ai reçu lors de mes visites dans son laboratoire.

J'ai aussi trouvé énormément d'encouragements de la part de beaucoup de mes collègues enseignants du département d'informatique, particulièrement tous les membres de l'équipe MePQoS, .

Je ne peux oublier ma femme pour son soutien, ses encouragements et son aide, sans elle, cette thèse n'aurait pas pu se concrétiser, qu'elle trouve ici mes remerciements et ma gratitude les plus profonds.

Résumé

Afin de pallier au problème de l'explosion combinatoire dont souffre les méthodes de vérification formelles, en particulier celles basées sur les modèles, plusieurs techniques ont été proposées. Nous nous intéressons dans cette thèse à l'approche basée sur la distribution. En effet, le fait de faire coopérer plusieurs stations permet d'offrir un espace mémoire et une puissance de calcul très importants, et permettre ainsi l'analyse de systèmes qui n'auraient jamais pu l'être sur une seule machine.

L'idée consiste alors à répartir l'espace d'états sur les différentes stations et d'adapter les algorithmes de vérification à ce nouveau contexte.

Dans la première partie de notre travail nous nous sommes intéressés à la génération distribuée de l'espace d'états et aux problèmes inhérents à celle-ci : équilibre de charges, reduction du temps d'oisiveté des processeurs, minimisation du nombre de messages échangés, etc. Nous avons proposé un certain nombre de solutions pour permettre de remédier à ces problèmes. Le choix d'une bonne fonction de hashage est déterminant, regroupement des messages, granularité, etc.

Dans une seconde étape, nous nous sommes intéressés aux propriétés dites générales : atteignabilité, vivacité, états d'accueil). Pour la première propriété on obtient de très bons résutats avec une accélération supra-lineaire. Les deuxième et troisième propriétés nécessitent la construction de toutes les composantes fortement connexes de l'espace, donc un nombre important d'échanges de messages, mais permet néanmoins d'analyser des problèmes de taille importante.

Nous nous sommes aussi interessés à la vérification de propriétés par le model-checking, nous avons axé notre travail sur la model-checking de CTL. Nous avons donné la démarche permettant de distribuer ces algorithmes et proposé de déterminer de manière distribuée le ou les contre-exemples lorsque la propriété à vérifiée n'est pas satisfaite.

Des prototypes ont été réalisés afin de mener des tests des algorithmes proposés.

Quand au model-checking de LTL nous nous sommes contentés de donner une proposition qui permettrait d'améliorer le recherche de cycles dans l'espace d'états.

Enfin, nous avons proposé une nouvelle approche de distribution de la vérification basée sur la modularité. Ainsi, chaque module du système est assigné à un processus client qui construit les graphes dits internes, le processus coordinateur construit le graphe de synchronisation.

Les algorithmes de vérification de l'atteignabilité, d'existence d'états bloquants, de la vivacité et détats d'accueil ont été aussi adaptés pour l'approche distribuée.

La conjugaison des approches modulaires et l'approche distribuée ont permis d'obtenir de très bons résultats, en particulier lorsqu'on a un système faiblement couplé. Ainsi, pour le problème des philosophes, par exemple, nous avons pu générer l'espace d'états d'un problème dont la taille normale est de plus de 6.10^{23}.

Table des matières

Introduction générale

Grâce aux progrès technologiques réalisés ces dernières décennies, les systèmes informatiques contrôlent de plus en plus des tâches dans notre entourage, de l'instrument médical au système de contrôle du trafic aérien. Il est évident que les systèmes qui assurent ces tâches, qu'ils soient matériels ou logiciels, doivent être fiables et sûrs. Ces systèmes sont souvent composés de plusieurs éléments pouvant s'exécuter en parallèle et communiquer entre eux. Leurs comportements sont définis par leurs interactions avec les événements internes ou externes et leur devenir dans le temps. Les protocoles de communication, les systèmes embarqués, les circuits logiques synchrones, les systèmes de production sont d'autres exemples de tels systèmes.

Une caractéristique importante de ces systèmes est leur propension à avoir un très grand nombre de comportements possibles. Cela est dû essentiellement à l'exécution concurrente et aux multiples interactions des différents éléments qui les composent. Cela rend aussi leur conception, leur réalisation et leur mise en œuvre extrêmement difficile. Cette difficulté est accrue par le rôle souvent critique de ce type de systèmes (processus de contrôle en avionique, centrales nucléaires, etc.), nous parlons alors de systèmes complexes.

Ces caractéristiques particulières font que les phases de vérification et de validation sont cruciales. Ces phases permettent de contrôler que le système satisfait bien les propriétés de fiabilité attendues. Il en résulte que tout problème lié au comportement qualitatif (sûreté, équité, absence de blocage, etc.), comme au comportement quantitatif (perte de messages, vitesse moyenne de transmission, etc.) doit être détecté et corrigé très tôt dans le cycle de développement.

Pour vérifier et prévenir ainsi d'éventuels dysfonctionnements, une pléthore de techniques ont été développées. Parmi celles-ci, les *tests* et la *simulation* [60] sont les plus connus et probablement les plus largement utilisés. Dans les vérifications basées sur les tests, le système à vérifier est testé pour révéler d'éventuelles erreurs sur un ensemble de situations choisies. Faute de temps et à cause des coûts engendrés, il est souvent impossible de tester ces systèmes pour toutes les situations possibles. L'ensemble des cas à tester doit

être choisi avec une attention particulière pour couvrir le maximum de scénarios distincts possibles. Néanmoins, il est impossible de garantir que le produit sera exempt de toute erreur.

Une autre technique pour assurer la validation de systèmes est celle basée sur l'utilisation de *méthodes formelles*. Pour pouvoir être réalisée, la vérification impose une description formelle du système, ainsi qu'une spécification formelle des propriétés. De nombreux formalismes de spécification dédiés aux systèmes concurrents ont été proposés dans la littérature tels que les automates communicants, systèmes de transition, algèbres de processus [33], CCS [50, 51], CSP [35], LOTOS, ESTELLE [65, 16], etc.

Parmi les formalismes les plus utilisés, les réseaux de Petri occupent une place prépondérante et se trouvent être l'un des outils ayant été le plus largement étudié, si l'on considère la diversité des techniques automatiques de vérification qui leurs sont associés [52, 26]. Ils offrent une structure très simple et constituent toujours le support de nombreuses études théoriques des systèmes concurrents et de leurs propriétés.

Les méthodes de vérification formelle sont basées sur les trois approches suivantes :
- Les vérifications basées sur les preuves de théorèmes.
- Les vérifications basées sur les équivalences.
- Les vérifications basées sur le modèle.

D'autres classifications sont évidemment proposées dans la littérature.

La première approche, basée sur les preuves de théorèmes [34], a pour principe de poser un ensemble d'axiomes, souvent donnés par le concepteur, puis à prouver un ensemble d'assertions déterminant ainsi la conformité du système. Cette preuve est faite plus ou moins manuellement. L'aide apportée par les outils tels que les démonstrateurs de théorèmes n'évite pas totalement la nécessité de l'intervention humaine. Ce type de vérification est rarement employé, il est utilisé essentiellement dans le domaine des spécifications algébriques et logiques [55].

La deuxième approche de vérification formelle consiste à vérifier l'équivalence (le plus souvent par rapport à une relation de bisimulation) entre le modèle de description de l'implémentation et une spécification qui décrit ce que l'on attend de cette implémentation [54], les deux utilisant le même formalisme de description. Si les deux modèles sont équivalents cela prouve que l'implémentation est conforme à la conception.

La dernière approche et la plus utilisée, basée sur les modèles, permet une vérification simple et efficace et elle est complètement automatisable. La vérification basée sur les modèles ou model-checking [28, 49] est surtout applicable pour les systèmes ayant un espace d'états fini. Les algorithmes de vérification associés à cette méthode utilisent l'ensemble des états que le système peut atteindre pour prouver la satisfaction ou la non-satisfaction des propriétés. Divers outils de vérification de systèmes existent. Parmi les plus utilisés, on peut citer SPIN [37] , Murφ [25], SVM [48], UPPAAL [7].

Cependant, le problème majeur de ce type de vérification est la taille souvent excessive de l'espace d'états. En effet, elle peut être exponentielle par rapport à la taille de la description du système. Une des causes principales de ce phénomène est le fait que l'exploration est réalisée en prenant en compte tous les entrelacements possibles d'événements concurrents.

Pour pallier au problème de l'explosion combinatoire, différentes solutions dont l'objectif est de réduire la taille de l'espace d'états ont été proposées dans la litérature. Elles sont généralement basées sur :

- L'utilisation de structures de données particulières.
- L'exploitation de l'ordre partiel sur les occurrences des événements.
- L'exploitation des symétries dans le système.
- L'exploitation de la modularité.

La première approche consiste à minimiser l'espace mémoire utilisé pour stocker l'espace d'états en choisissant une représentation des états et des comportements du systèmes particulièrement concise [15]. Les diagrammes de décisions binaires (BDD : Binary Decision Diagram) sont une structure de données permettant la représentation de fonction booléennes sur laquelle les opérations booléennes classiques peuvent être réalisées efficacement. En codant d'une part l'ensemble des états du système et d'autre part la relation de transition entre les états, sous forme de diagrammes de décision binaires, il est alors possible de définir des outils de vérification pour un grand nombre de propriétés. Cette approche est utilisée dans plusieurs outils de vérification tels que SMV, Uppaal ou encore prism. D'autres structures basées sur un principe similaire ont été définies, telles que les diagrammes de décision de données (DDD : Data Decision Diagram) ou encore les diagrammes de décision hiérarchiques (SDD : Set Decision Diagram) [15].

La seconde approche permettant de pallier à l'explosion combinatoire du nombre d'états est la prise en compte de la concurrence lors de la construction du graphe d'état. En effet, les différents entrelacements possibles d'événements concurrents induisent un grand nombre de séquences composées d'un même ensemble de transitions, partant et arrivant aux mêmes états où seul l'ordre d'apparition diffère. Ces séquences passent par un nombre important d'état différents. L'idée est de réduire le graphe de comportement grâce à l'équivalence de séquences [53, 66].

Une autre approche est basée sur le fait que certains systèmes concurrents sont constitués d'éléments ayant des comportements similaires et donc comportant des symétries, i.e., certains états sont identiques à des permutations près [3, 5, 22]. Des techniques de factorisation permettent la construction de graphes, dans lesquels un nœud ne représente pas qu'un état, mais un ensemble d'états équivalents du point de vue de ces symétries. On obtient ainsi un graphe réduit qui peut être utilisé pour vérifier des propriétés. Ces propriétés seront dites elles aussi propriétés symétriques.

La dernière approche est basée sur l'exploitation de la modularité dans les systèmes [18, 32, 70]. En effet, dans certains systèmes, la vérification de certaines propriétés du système

global, peut être réduite à la vérification de propriétés locales dans chaque module. Ainsi, au lieu de construire l'espace d'états du système global, on construit ceux des modules, dont la taille est beaucoup plus réduite. A partir des propriétés des différents modules, on peut parfois déduire les propriétés du système global.

Le point commun entre toutes ces démarches, qui ont montré leur efficacité, consiste en la réduction de l'espace d'états, en réduisant soit le nombre d'états, soit la taille occupée en améliorant la représentation de cet espace d'états. Néanmoins, l'expérience pratique a montré qu'aucune de ces techniques prises séparément ne peut suffire pour éviter l'explosion d'états dans tous les cas de figure, et leur utilisation conjointe a de plus grandes chances d'être plus efficace.

Il existe une autre démarche qui consiste plutôt à offrir un espace de stockage et une puissance de calcul plus importants. Cette démarche est basée sur l'idée d'exploiter la puissance de calcul et la capacité mémoire fournies par un ensemble de machines ayant chacune sa propre mémoire, connectées en réseau et communiquant par l'envoi de messages. Cette approche est justifiée surtout dans un contexte ou les réseaux d'interconnexion sont de plus en plus disponibles et performants. L'idée est alors de répartir l'espace d'états dans un environnement distribué. Il faut alors adapter les algorithmes de vérification pour tirer parti au mieux de cette architecture particulière.

La génération parallèle et distribuée de l'espace d'états a été étudiée dans des contextes variés et plusieurs solutions ont été proposées. La majorité de ces solutions adoptent une approche commune : Le style de programmation est SPMD (Single Program Multiple Data). Ils peuvent être considérés comme étant des algorithmes parallèles homogènes, dans le sens où les processus exécutent le même code.

L'idée d'utiliser un environnement distribué dans la vérification formelle n'est pas complètement nouvelle. Plusieurs travaux se sont intéressés à cette approche. Dans [62], les auteurs ont donné une description d'une version parallèle de $Mur\varphi$. L'ensemble de tous les états visités est réparti sur les nœuds d'une machine virtuelle parallèle. L'énumération explicite des états est effectuée en parallèle. Une approche similaire a été proposée pour SPIN [36] dans [6], mais avec une différence dans la manière dont sont répartis les états. Une version distribuée d'UPPAAL basée sur le même principe que la version parallèle de $Mur\varphi$ a été proposée dans [62]. Les algorithmes distribués proposés dans ces travaux concernent principalement des problèmes d'atteignabilité. Dans le même sens, les travaux de H. Garavel, R. Mateescu et I. Smarandache [31] traitent de la construction parallèle d'un espace d'états réparti d'un système de transitions étiqueté.

plusieurs travaux concernant la distribution des algorithmes de Model-checking ont été réalisés. Ces travaux proposent des versions parallèles et distribuées pour le Model-checking LTL. Le travail de F. Lerda et R. Sisto présente la version distribué du Model-Checker

SPIN [45, 6]. La distribution du Model-checking CTL est étudiée dans [38].
Dans [40], une approche distribuée basée sur les Systèmes d'Equations Booléennes (SEBs), qui fournissent une représentation intermédiaire des problèmes de vérification définis sur des Systèmes de Transitions Etiquetées, est utilisée afin d'augmenter les capacités de vérification.

L'approche basée sur des machines parallèles, où plusieurs processeurs partagent une même mémoire, a aussi suscité l'intérêt des chercheurs, même si les travaux sont relativement rares [58]. Cette disparité peut être expliquée par le coût relativement élevé de ce type de machines. Les principaux avantages des machines parallèles est l'utilisation concurente d'une mémoire partagée, évitant ainsi le recours à la communication par envoi de messages, de plus il n'est plus besoin de partitionner l'espace d'état. Néanmoins, certaines difficultées liées à cette approche surgissent. Ainsi, il est nécessaire de synchroniser les opérations de manipulation de données partagées afin d'assurer leur consistance. Des techniques de synchronisation permettant d'assurer l'exclusion mutuelle doivent être utilisées, tout en essayant de garantir un degré de parallélisme élevé.
Des travaux récents s'intéressent aussi à vérification parallèle, en exploitant la puissance de calcul des processeurs multicøeurs [29], utilisant une mémoire partagée.

Les architectures distribuées les plus utilisées ne disposent pas d'une mémoire partagée et sont très génériques, car composées essentiellement de machines interconnectées par un réseau standard tel que ceux utilisés habituellement dans les entreprises et laboratoires académiques. Ce type d'architecture est plus couramment appelée grappe de stations de travail (pouvant être par exemple une grappe de PCs). Cette architecture implique généralement un mécanisme de distribution des tâches et des données par échange de messages qui nécessite un système de gestion efficace des communications engendrées par le calcul entre les noeuds.

Dans cette thèse, nous nous intéressons aussi à la distribution de la vérification comme moyen pour augmenter la taille des systèmes à vérifier. Nous commençons par donner des algorithmes de génération de l'espace d'états distribué et notre première contribution consiste à définir des algorithmes de vérification sur un espace d'états distribué de propriétés qualitatives, propriétés dites générales [11], telles que la vivacité et l'existence d'état d'accueil, basées sur le calcul des composantes fortement connexes, et de propriétés d'atteignabilités telles que l'existence d'états de blocage qui peuvent être vérifiées pendant la génération.

Les performances de la vérification distribuée dépendent largement de la fonction de répartition. Le choix de cette fonction doit tant que possible permettre un équilibre de charge, et minimiser le temps d'oisiveté des processus. Dans cette optique, nous avons utilisé Charm++ et AMPI (Adaptive Message Passing Interface) pour mettre en œuvre ces

techniques d'équilibrage de charge basées sur la virtualisation [56].

Nous nous penchons aussi sur la vérification distribuée de propriétés dites spécifiques, exprimées dans une logique temporelle. Il y a beaucoup de travaux dans ce sens mais la majorité sont orientés vers la logique temporelle lineaire LTL, Nous avons donc voulu axé notre travail sur la distribution du model checking de CTL. Nous avons aussi proposé la recherche distribuée de contres exemples dans le cas où la propriété n'est pas valide [13].

Enfin, nous proposons une nouvelle approche pour la vérification distribuée basée sur la modularité. Dans cette approche, les systèmes considérés sont constitués d'un ensemble de modules qui synchronisent à travers des transitions communes. On associe chacun des modules à un processus qui se charge de la génération de l'espace d'états local. Un processus coordinateur se charge de la construction du graphe de synchronisation. Des algorithmes de vérification de propriétés, basés sur l'espace d'états modulaire distribué sont aussi proposés [12].
L'ensemble des algorithmes proposés dans cette thèse ont été implémentés et testés.

Cette thèse est organisée comme suit :

Dans le chapitre 1 seront présentées les techniques de vérification et les modèles de spécification classiques utilisés pour l'analyse des systèmes complexes.

Le chapitre 2 sera consacré à la description des algorithmes distribués de la génération de l'espace d'états. Le rôle et les types des fonctions de hashage y sont étudiés ainsi que leur influence sur l'équilibre de charge et les performances des algorithmes distribués.

Dans le chapitre 3 sont présentés les algorithmes distribués pour la vérification de propriétés générales appliquées aux réseaux de Petri telles que le blocage, la vivacité et les états d'accueil.

Le chapitre 4 présente la distribution du model-checking pour la logique temporelle linéaire LTL. Il est mis en évidence les difficultés rencontrées pour l'adaptation des algorithmes NDFS, basés sur un double parcours en profondeur d'abord pour la détection de cycles d'acceptation.

Au chapitre 5, le model-checking distribué pour la logique temporelle arborescente CTL est présenté de manière détaillée.

Le chapitre 6 est consacré à la vérification distribuée des systèmes modulaires. Chaque processus prend en charge la génération de l'espace interne d'un module et coopère avec le processus coordonateur pour la construction du graphe de synchronisation. Tous les processus coopèrent aussi pour la vérification de propriétés.

Enfin, nous terminons cette thèse par une conclusion et quelques perspectives de recherche.

Chapitre 1

Techniques de vérification des systèmes

Les systèmes à concevoir deviennent de plus en plus complexes et critiques. Ceci implique la nécessité d'une validation formelle avant leur implémentation. L'utilisation des techniques de vérification formelle, pour prouver la correction des systèmes à construire, est donc une approche nécessaire pour concevoir des systèmes corrects et valides.
Dans ce chapitre, nous commencerons par décrire les différentes techniques de vérification, leurs domaines d'application et leur efficacité pour la validation des systèmes. Enfin, nous donnerons un bref aperçu de quelques outils et formalismes utilisés dans le cadre de la vérification formelle.

1.1 Spécification de systèmes et modèles

L'approche adoptée le plus souvent est celle orientée modèle, très connue dans la plupart des disciplines d'ingénierie. Au lieu de construire un objet directement, on conçoit d'abord un modèle, lequel est analysé et vérifié entièrement. Une fois que le modèle obtenu est satisfaisant, l'objet peut vraiment être construit.

En ce qui concerne les systèmes informatiques de grande taille, le modèle est une abstraction (ou simplification) du comportement de parties critiques de ce système, et la vérification consiste à prouver que le modèle satisfait toutes les propriétés désirées. Les recherches en méthodes formelles ont permis de définir un ensemble complet de notations pour décrire les systèmes étudiés, et pour spécifier toutes les propriétés intéressantes, incluant les propriétés comportementales et temporelles. Il s'agit alors de prouver que le système satisfait les propriétés requises.

1.1.1 Spécification opérationnelle

La spécification opérationnelle consiste à décrire le système à analyser par un modèle en utilisant un langage ou un formalisme de description. Les algèbres de processus [51], les réseaux de Petri [52], les automates et les systèmes d'événements [68] sont des exemples de formalismes que l'on peut utiliser.

Le modèle du système est constitué d'un ensemble d'états initiaux, et d'un ensemble de transitions permettant de définir l'ensemble des états atteignables par application des règles d'évolution du formalisme de description.
A partir de la spécification opérationnelle, on déduit un graphe d'accessibilité qui décrit tous les comportements du système (l'ensemble des chemins du graphe). Un comportement est un chemin fini ou infini.
Les modèles mathématiques qui sont généralement utilisés pour définir précisément l'ensemble des chemins représentés par le graphe sont les systèmes de transitions étiquetées.

1.1.2 Spécification descriptive de propriétés

Les propriétés d'un système peuvent être divisées en deux groupes, les propriétés statiques et les propriétés dynamiques.
Les propriétés statiques telles que les invariants peuvent être exprimées en utilisant la logique des prédicats du 1^{er} ordre et la logique de Hoare [34] par exemple. Par contre, ce n'est pas le cas pour les propriétés dynamiques, c'est-à-dire les propriétés sur les comportements des systèmes modélisés. Les comportements sont les chemins d'exécution des systèmes de transitions. Les propriétés sur ces derniers peuvent être exprimées à l'aide de la logique temporelle (Temporal Logic) [21, 49]. Il existe plusieurs langages de logique temporelle, les plus utilisés sont la logique temporelle linéaire LTL (Linear Temporal Logic), la logique temporelle arborescente CTL (Branching Temporal Logic) et plus particulièrement la logique CTL* [59].

1.2 Vérification

A cause de leur criticité, la correction des systèmes complexes doit être garantie par une analyse fine et exhaustive de leurs comportements (leurs modèles opérationnels) [8]. La vérification est le seul moyen qui assure cette analyse et ainsi, une validité des systèmes que l'on construit.
Qui dit vérification, dit comparaison de deux énoncés. Le premier décrit le **quoi**, on l'appelle Spécification Descriptive (au sens où il décrit ce que l'on doit faire sans rien dire du comment faire).
Le second décrit le **comment**, on l'appelle donc Algorithme ou Spécification Opérationnelle

(au sens où il décrit comment opérer pour résoudre le problème).
Le but de la vérification est de s'assurer que le second réalise bien ce que décrit le premier. La vérification consiste donc, à assurer un fonctionnement correct du système par rapport aux spécifications vues comme un ensemble de propriétés.
Une méthode de vérification est fondée sur :

1. Un langage de description d'une spécification descriptive.
2. Un langage de description d'une spécification opérationnelle.
3. Un algorithme de vérification qui est une procédure de décision permettant d'établir si les deux énoncés sont consistants.

La démarche « **spécifier pour vérifier** » est une approche qui propose de décrire le système par un modèle, de spécifier les besoins et les propriétés puis de vérifier si le modèle du système (la solution proposée) a les propriétés attendues. La figure 1.1 illustre cette démarche.

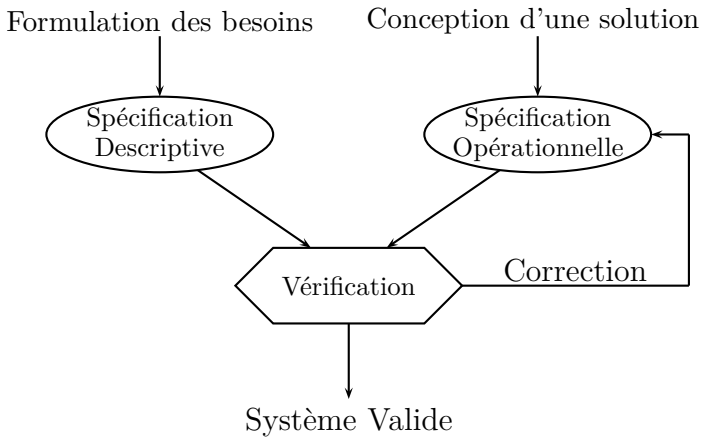

FIGURE 1.1 – Illustration de la démarche « spécifier pour vérifier »

Dans ce qui suit, nous présentons les moyens permettant l'élaboration de la vérification. Nous explorons ainsi les différents types de spécification et les différentes méthodes utilisées pour la vérification des systèmes.

1.3 Vérification syntaxique et vérification sémantique

Concernant les moyens de vérification, nous distinguons deux approches :
- L'approche des méthodes syntaxiques qui regroupent les techniques basées sur la démonstration de théorèmes [34].
- L'approche des méthodes sémantiques qui regroupent les méthodes de model-checking et les techniques de simulation. Elle sont dites sémantiques, car elles reposent sur l'analyse du modèle opérationnel (sémantique associée à la syntaxe), par exemple les automates [28].

1.3.1 Les méthodes de preuve

Les méthodes de preuve, ou méthodes syntaxiques, sont basées sur la démonstration de théorèmes. Elles utilisent soit la logique de premier ordre, soit la logique d'ordre supérieur, la relation entre la spécification et l'implémentation est un théorème à démontrer dans une logique. L'implémentation fournit les axiomes et les hypothèses pour la preuve. La démonstration est basée sur la déduction logique, la réécriture et le mécanisme de la preuve par récurrence. Cette technique est très puissante pour les descriptions matérielles de haut niveau, car la taille des données n'a plus d'importance. Elle s'applique à tous les niveaux d'abstraction. Elle est capable de traiter des spécifications non bornées et de prouver des propriétés algorithmiques complexes. Ces techniques se caractérisent par les propriétés suivantes :
- elles sont basées sur les démonstrations de théorèmes. Elles fonctionnent assez bien pour des propriétés invariantes et plus généralement de sûreté ;
- elles permettent de traiter des systèmes infinis ;
- elles ne sont pas entièrement automatiques ;
- le processus de vérification n'est que semi-décidable ;
- de plus, elles ne sont pas faciles à utiliser : demandent du temps et de l'expertise humaine.

1.3.2 La simulation

Les simulations [60] reposent sur la mise en œuvre de modèles théoriques qui servent à étudier le fonctionnement et les propriétés d'un système modélisé ainsi qu'à en prédire son évolution. Cependant, les techniques de simulation posent les problèmes suivants :
- Le problème de la complétion de la procédure de vérification (exploration de certains comportements du système seulement).

– Le temps et l'espace mémoire important nécessaires pour la réalisation de la simulation.

Les procédures de simulation peuvent être utilisées pour déceler un ensemble important d'erreurs, mais restent néanmoins inefficaces dans certains cas, en particulier pour les systèmes critiques. En effet, ces systèmes sont caractérisés par la nécessité d'assurer un fonctionnement sans faille. Il est clair que les techniques de vérification basées sur la simulation ne couvrent pas tous les scénarios possibles. Ils ne garantissent donc pas la correction totale des systèmes testés.

1.3.3 La vérification formelle basée sur les modèles

C'est une technique de vérification exhaustive appelée model-checking. Elle assure la correction de tous les comportements du système vis-à-vis de la propriété vérifiée. Cette technique est complètement automatique et est caractérisée par les propriétés suivantes :

– Possibilité de montrer des contre-exemples : dans le cas où la propriété n'est pas vérifiée, les approches model-checking permettent de générer des traces d'exécutions qui invalident la propriété.
– Utilisation de logiques temporelles : le modèle offre la possibilité de spécifier les propriétés en logique temporelle. Ce qui permet d'exprimer plusieurs propriétés de fonctionnement du système.
– En général, les systèmes réels ont une grande taille. La vérification par les modèles pose alors le problème de l'explosion combinatoire de l'espace d'états.

La vérification des propriétés s'effectue généralement par une énumération exhaustive des états accessibles. L'efficacité de cette technique dépend en général de la taille de l'espace des états accessibles et trouve donc ses limites dans les ressources (moyens techniques et matériels) mises en œuvre pour traiter l'ensemble de ces états.

1.4 Les formalismes et outils les plus utilisés

Notre objectif ici n'est pas de faire une présentation exhaustive de ces méthodes, mais de présenter quelques exemples de langages de spécification opérationnelle, quelques outils permettant la vérification des systèmes et quelques langages de spécification de propriétés.

En ce qui concerne les méthodes de spécification opérationnelle, la méthode B [1], les automates [68], le formalisme des réseaux de Petri [52] et le formalisme des algèbres de processus [34, 51], sont les moyens de description opérationnelle les plus connus et les plus utilisés.

Pour chacun de ces formalismes, de nombreux outils ont été implémentés afin de permettre la spécification de systèmes et leur vérification, et parmi les outils les plus connus on trouve :

- **SPIN** (Simple Promela INterpreter) [37] qui est un outil open source de modélisation et de vérification de systèmes distribués comme les protocoles de communication, les applications multi-thread, etc. Les systèmes sont décrits dans le langage Promela par un ensemble d'automates communicants et les propriétés sont exprimées par des formules de la logique temporelle linéaire. SPIN est considéré comme étant l'un des outils de vérification les plus utilisés.
- **UPPAAL** [7] est un outil de vérification des systèmes temps-réel, les systèmes sont modélisés à l'aide d'un ensemble d'automates temporisés. L'outil vérifie si le système satisfait un ensemble de proprétés invariantes.
- **Tina** est aussi un outil de vérification de systèmes temps-réel utilisant les réseaux de Petri temporisés. L'espace d'états peut tre réduit en utilisant une construction symbolique en considérant des classes d'équivalence de l'espace d'états. Les auteurs Vernadat et Berthomieu [9] ont défini plusieurs abstractions qui préservent différentes classes de propriétés.
- **Mur-ϕ** [25] est un outil formel de vérification basé sur l'énumération explicite des états. Le langage de description des systèmes est basé sur la notion de commandes gardées (condition/action). Mur-ϕ utilise un ensemble de techniques telles que la symétrie afin de limiter l'explosion de l'espace d'états.
- **SMV** (Symbolic Model Verifier) [48] est un outil de vérification de systèmes matériels, qui a été développé par l'équipe d'Edmund Clarke à l'université de Carnegie Mellon. Depuis 1999, les efforts de développement se sont concentrés sur MuSMV qui est une extension de SMV. Ce dernier est aussi basé sur la vérification symbolique et inclut le Model-checker SAT. Cet outil qui utilise les BDD, permet de vérifier des systèmes à états finis avec des spécifications CTL.
- **PROD** est un outil développé à l'Université de Technologie d'Helsinki. Il permet la génération de l'espace d'états et la vérification de propriétés LTL et CTL. PROD est utilisé comme model-checker pour LOTOS.
- **CPN-Tools** [39] est un outil graphique permettant l'édition, la simulation et l'analyse des réseaux de Petri colorés. CPN Tools a été développé à l'université Aarhus par Kurt Jensen, Sren Christensen, Lars M. Kristensen et Michael Westergaard. Actuellement, CPN Tools continue d'tre developpé à l'université d'Eindhoven.

La liste n'est pas exhaustive, plusieurs autres outils existent, à titre d'exemple, pour le langage de spécification B, l'outil de preuve appelé l'atelier B a été développé [57] ; GreatSPN pour la vérification quantitative (analyse des performances) des systèmes, ...

Dans les méthodes de vérification des systèmes, la spécification des propriétés constitue une étape cruciale, il s'agit essentiellement de poser les bonnes questions et de les exprimer dans un langage de spécification adéquat. Les propriétés décrivent donc les comportements souhaités du système.

1.5 Classification des propriétés

Selon le type de système à vérifier, et les besoins attendus par les utilisateurs, les propriétés à vérifier sont classées en deux grandes catégories :

- Les propriétés qualitatives.
- Les propriétés quantitatives.

Les premières concernent le fonctionnement du système à vérifier. Elles sont exprimables dans plusieurs modèles comme les systèmes d'événements, les automates et les structures de Kripke [28]. Généralement, elles s'expriment à l'aide de logiques temporelles.

Les secondes recouvrent des questions de performances comme la durée maximale d'attente d'un service ou le temps de réponse (par exemple « *la barrière sera abaissée moins de 5 secondes après le signal d'arrivée du train* »).

1.5.1 Propriétés qualitatives

Les propriétés qualitatives peuvent être classées en cinq grandes catégories :

La sûreté : Cette propriété énonce que, sous certaines conditions, quelque chose « de mauvais » ne doit pas se produire (« ne se produit jamais »).

La vivacité : énonce que, sous certaines conditions, quelque chose « de bon » finira par avoir lieu.

L'atteignabilité : L'atteignabilité énonce qu'une certaine situation peut être atteinte.

L'absence de blocage : L'absence de blocage indique que le système ne se trouve jamais dans une situation où il ne peut plus progresser. Un état bloquant (deadlock state) d'un système est un état à partir duquel aucune action ne peut être effectuée. Une propriété importante que l'on souhaite souvent assurer dans un système est l'absence de blocage. Il est à noter qu'il est nécessaire de distinguer entre les états bloquants (blocage non souhaité) et les états de terminaison (états finaux). Ainsi, nous pouvons dire que : dans l'espace d'états

caractérisant un système, un état qui n'est source d'aucune transition peut correspondre à une des deux situations suivantes :

- Terminaison : dans ce cas, l'état correspond à un état final du système. Cette propriété caractérise une terminaison normale du système.
- Blocage : cette propriété décrit un comportement qui met le système dans un état où il ne peut évoluer.

L'équité (fairness) : Une autre propriété importante qui peut caractériser l'exécution d'un système est l'équité. Elle assure que lorsqu'il y a, dans un algorithme, des choix non déterministes, certains choix possibles ne seront pas délaissés (problème appelé aussi famine). Le principe des preuves d'équité est appliqué, par exemple, à l'étude des ascenseurs (toutes les requêtes des usagers finissent par être satisfaites).

Les propriétés autres que celles de sêreté ou de vivacité peuvent se ramener à la conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Une autre classification, consiste à regrouper les propriétés selon leur procédure de vérification. Dans cette classification, on distingue deux classes : les propriétés comportementales et les propriétés dynamiques.

1.5.2 Propriétés comportementales

Elles expriment un comportement attendu du système observé à un certain niveau d'abstraction. Une propriété comportementale peut être modélisée par un système de transitions étiquetées. La vérification consiste à comparer le système et la propriété à l'aide d'une relation d'équivalence ou de pré-ordre [51].

1.5.3 Propriétés logiques ou dynamiques

Elles décrivent une propriété globale du système, comme l'absence d'interblocage, l'exclusion mutuelle, etc. Les logiques temporelles sont des formalismes bien adaptés pour exprimer ces propriétés. La vérification consiste à montrer que le système satisfait la formule logique qui exprime la propriété.

1.6 Spécification par logiques temporelles

Les logiques temporelles permettent d'exprimer des propriétés sur les états et les transitions d'un système. Ces logiques sont généralement partagées en deux classes :

– **Les logiques du temps linéaire** : comme LTL, permettent d'exprimer des propriétés portant sur des séquences d'exécution ou comportements linéaires (issus de l'état initial) du système. La figure 1.2 illustre une séquence d'exécution.

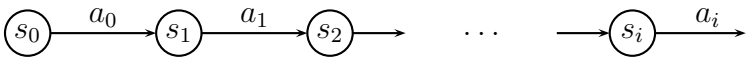

FIGURE 1.2 – Une squence d'excution

– **Les logiques du temps arborescent** : comme CTL et CTL*, permettant d'exprimer des propriétés portant sur les arbres d'exécution ou comportements arborescents (issus de l'état initial) du système. La figure 1.3 représente un comportement arborescent.

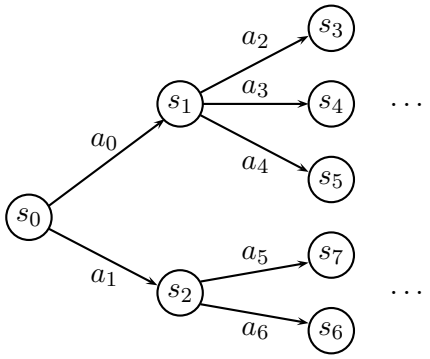

FIGURE 1.3 – Un arbre d'excution

Pnueli a été le premier à introduire les notions sur la logique temporelle [55], en définissant des opérateurs traduisant les propriétés que l'on peut exprimer en langage naturel. Par exemple, `toujours`, `suivant`, `inévitablement`, `jusqu'à`, etc. Ceci rend relativement simple la formalisation d'une propriété en logique temporelle. Ainsi, Penueli a défini la logique temporelle linéaire LTL qui permet l'expression des propriétés sur les comportements linéaires d'un système.
Par la suite, la logique temporelle arborescente CTL a été introduite par Clarke et Emerson [4] en 1980. Cette logique permet d'exprimer des propriétés sur des arbres d'exécution.

1.6.1 La logique temporelle CTL*

La logique temporelle CTL* permet d'exprimer des propriétés sur les chemins aussi bien que sur les arbres d'exécution. Suivant la présentation synthétique de [28], nous donnons la définition de CTL* et nous précisons ensuite ses fragments purement linéaires (LTL) et purement arborescents (CTL).

Soit AP un ensemble de propositions atomiques ; et soit p un élément de cet ensemble. La syntaxe de CTL* est définie comme suit :

Définition 1 (Syntaxe de CTL*) *Soient φ et ψ deux formules temporelles quelconques et $p \in AP$ une proposition atomique.*
Les formules CTL sont définies ainsi :*

- *Toute proposition atomique $p \in AP$ est une formule temporelle,*
- *$\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ et $\varphi \Rightarrow \psi$ sont des formules temporelles,*
- *$X\varphi$ et $\varphi U \psi$, sont des formules temporelles,*
- *$E\varphi$ et $A\varphi$ sont des formules temporelles.*

Les symboles X et U sont des opérateurs temporels dont la sémantique sera présentée ultérieurement.

Structure de Kripke

Les modèles de la logique temporelle sont définis sur des automates appelés structure de Kripke [21]. On définit une structure de Kripke comme suit :

Définition 2 (Structure de Kripke) *Une structure de Kripke est un quadruplet $K = \langle Q, Q_0, T, \lambda \rangle$ où :*

- *Q est l'ensemble des états du système,*
- *Q_0 est l'ensemble des états initiaux,*
- *T est l'ensemble de transitions inclus dans $Q \times Q$,*

- *λ est l'interprétation des variables sur les états sous forme d'une application de Q dans 2^{AP}, où AP est un ensemble de propositions atomiques. Cette application associe à tout état q l'ensemble des propositions vraies dans cet état.*

Une séquence σ est une suite infinie d'états $\sigma = s_0 s_1 s_2 \ldots \in Q^\omega$ telle que : $\forall i \geq 0$ $(s_i, s_{i+1}) \in T$.

On note par $\sigma(i)$ l'élément i (l'état i) du chemin σ.

La séquence finie $s_0 s_1 \ldots s_n$ est dite préfixe de σ, et notée $\sigma[0..n]$ si $\sigma(i) = s_i, \forall 0 \leq i \leq n$, .

On désigne par $Run(M)$ l'ensemble des exécutions possibles dans le modèle M.

Soit μ une séquence dans le modèle M, la formule $M, \mu, i \models \varphi$ se lit ainsi : à partir de l'état $\mu(i)$ de la séquence d'exécution μ dans le modèle M, la propriété φ est satisfaite.

Dans la suite, on omettra M car le contexte est implicite.

Définition 3 (Sémantique de CTL*) *Soient φ et ψ deux formules temporelles quelconques et $p \in AP$ une proposition atomique. La sémantique de CTL* est définie inductivement de la façon suivante :*

- $\mu, i \models p \Leftrightarrow p \in \lambda(\mu(i))$
- $\mu, i \models \neg\varphi \Leftrightarrow \mu, i \not\models \varphi$
- $\mu, i \models \varphi \vee \psi \Leftrightarrow \mu, i \models \varphi$ *ou* $\mu, i \models \psi$
- $\mu, i \models \varphi \wedge \psi \Leftrightarrow \mu, i \models \varphi$ *et* $\mu, i \models \psi$
- $\mu, i \models X\varphi \Leftrightarrow \mu, i+1 \models \varphi$
- $\mu, i \models \varphi U \psi \Leftrightarrow \exists j, i \leq j \Rightarrow \mu, j \models \psi$ *et* $\forall k, i \leq k < j \Rightarrow \mu, k \models \varphi)$
- $\mu, i \models A\varphi \Leftrightarrow \forall \rho, (\rho \in Run(K)$ *et* $\mu[0..i] = \rho[0..i]) \Rightarrow \rho, i \models \varphi$
- $\mu, i \models E\varphi \Leftrightarrow \exists \rho, (\rho \in Run(K)$ *et* $\mu[0..i] = \rho[0..i]) \Rightarrow \rho, i \models \varphi$

Le modèle M satisfait la propriété φ, noté $M \models \varphi$, si et seulement si $\mu, 0 \models \varphi$ pour tout chemin μ de M.

1.6.2 LTL : Logique temporelle linéaire

La logique temporelle linéaire [55] permet de spécifier des propriétés d'exécutions sur les comportements linéaires. La logique temporelle LTL est un fragment de CTL* sans les quantificateurs A et E. La syntaxe de LTL est donc définie comme suit :

Définition 4 (Syntaxe de LTL) *Soit un ensemble fini de propriétés atomiques AP, et soient les constantes booléennes T (`true`) et F (`false`). Les formules de la logique LTL sont définies ainsi :*

- $T, F \in LTL$,
- $p \in AP \Rightarrow p \in LTL$
- $\varphi, \psi \in LTL \Rightarrow \varphi \wedge \psi, \varphi \vee \psi, \neg\varphi \in LTL$
- $\varphi, \psi \in LTL \Rightarrow \varphi U \psi, X\varphi \in LTL$

La sémantique de LTL est définie par rapport à des exécutions infinies σ dans une structure de Kripke.

Définition 5 (Sémantique de LTL) *Soient $\rho = s_1 s_2 \ldots$ une séquence d'états, i un entier tel que $i \geq 1$ et ϕ une formule LTL. La relation $\rho, i \models \phi$ (la séquence ρ satisfait ϕ à l'état s_i) est définie comme suit :*

- $\rho, i \models T$
- $\rho, i \models p \Leftrightarrow p \in s_i$
- $\rho, i \models \neg p \Leftrightarrow p \notin s_i$
- $\rho, i \models \phi \vee \varphi \Leftrightarrow \rho, i \models \phi \vee \rho, i \models \varphi$
- $\rho, i \models X\phi \Leftrightarrow (i < |\rho| \Rightarrow \rho, i+1 \models \phi)$
- $\rho, i \models \phi U \varphi \Leftrightarrow (\exists j \geq i$ *tel que* $\rho, j \models \varphi$ *et* $\forall i \leq k < j, \rho, k \models \phi)$.

On note $\rho \models \phi$ au lieu de $\rho, 0 \models \phi$.

Les opérateurs F et G

Pour la simplification des écritures des formules LTL, nous avons les opérateurs F et G suivants :

- $F\phi =^{def} T\,U\phi$ (éventuellement ϕ)
- $G\phi =^{def} \neg F\neg\phi$ (toujours ϕ).

La logique LTL permet donc d'énoncer des propriétés sur les traces des comportements d'un système. Elle permet d'exprimer l'évolution d'un système au cours du temps en examinant la suite des événements observés lorsque le système réalise un comportement linéaire. Les formules sont construites à partir des propriétés élémentaires, des opérateurs et constantes booléennes et des deux opérateurs temporels : l'opérateur X (qui se lit neXt) et l'opérateur binaire U (qui se lit Until).

Dans le cadre des logiques temporelles linéaires le temps se déroule linéairement. En clair, on spécifie le comportement attendu d'un système sans réelle possibilité de spécifier plusieurs futurs possibles à chaque instant. Dans la section suivante, nous présentons la logique temporelle arborescente. Cette logique permet de spécifier plusieurs futurs possibles.

Exemples de formules LTL

1. GFp : Toujours il arrivera que p soit vérifiée, c'est-à-dire que la proposition p est vraie infiniment souvent dans le cadre des mots infinis.
2. $G(r \Rightarrow Fg)$: exprime la vivacité ; Toujours, une requête (r) est suivie ultérieurement d'une réponse (g).
3. $(GFp_1 \wedge \ldots \wedge GFp_n) \Rightarrow G(r \Rightarrow Fg)$ C'est une combinaison des équations précédentes : si les événements $p_1, p_2, ..., p_n$ sont tous vrais infiniment souvent, alors on exige que toute requête soit suivie d'une réponse. Les p_i pourraient représenter ici des conditions nécessaires pour que le système soit en état de fonctionnement correct et puisse garantir une réponse à toute requête.

1.6.3 CTL : Computation Tree Logic

La logique temporelle CTL a été introduite par Clarke et Emerson [4] en 1980. Formellement, la sémantique de cette logique peut être définie pour les structures de Kripke et pour les systèmes de transitions. Elle permet d'exprimer des propriétés aussi bien sur des chemins que sur les arbres d'exécutions, c'est pour cette raison qu'elle est classée comme étant une logique temporelle arborescente, elle est basée sur des modèles où à tout moment il peut y avoir plusieurs futurs possibles. La syntaxe des formules CTL est donnée dans la définition suivante.

Définition 6 *(Syntaxe de CTL)*

- $p \in AP$ *est une formule de CTL,*
- *Si* φ *et* ψ *sont des formules de CTL, alors* $\neg\varphi$, $\varphi \wedge \psi$, *et* $\varphi \vee \psi$ *sont des formules temporelles de CTL,*
- *Si* φ *et* ψ *sont des formules de CTL, alors* $EX\varphi$, $AX\varphi$, $E(\varphi U \psi)$ *et* $A(\varphi U \psi)$ *sont des formules temporelles de CTL,*

Les symboles E et A sont des quantificateurs universels. Ils expriment respectivement que la propriété est valide sur au moins un chemin ou pour tout chemin. Les opérateurs X et U sont des opérateurs temporels, appelés respectivement $Next$ et $Until$. Ils doivent se trouver dans le champ d'un opérateur existentiel ou universel.

Définition 7 *(Semantique de CTL)*
Soit $p \in AP$ *une proposition atomique,* $M = (S, T, \lambda, s_0)$ *une structure de Kripke,* $s \in S$, σ *une séquence infinie de* M, φ *et* ψ *des formules CTL. La relation de satisfaction* $\models$ *est définie inductivement par :*

$s \models p$	$ssi\ \ p \in \lambda(s)$
$s \models \neg\varphi$	$ssi\ \ \neg(s \models \varphi)$
$s \models \varphi \vee \psi$	$ssi\ \ (s \models \varphi) \vee (s \models \psi)$
$s \models EX\varphi$	$ssi\ \ \exists\sigma$ *telle que* $\sigma(0) = s,\ \sigma(1) \models \varphi$
$s \models AX\varphi$	$ssi\ \ \forall\sigma$ *telle que* $\sigma(0) = s,\ \sigma(1) \models \varphi$
$s \models E[\varphi \cup \psi]$	$ssi\ \ \exists\sigma$ *telle que* $\sigma(0) = s\ \ et\ \ \exists j \geq 0,\ \sigma(j) \models \psi\ \wedge(\forall\, 0 \leq k < j,\ \sigma(k) \models \varphi)$
$s \models A[\varphi \cup \psi]$	$ssi\ \ \forall\sigma$ *telle que* $\sigma(0) = s\ \ et\ \ \exists j \geq 0,\ \sigma(j) \models \psi\ \wedge(\forall\, 0 \leq k < j,\ \sigma(k) \models \varphi)$

De même que pour LTL, des abreviations sont aussi définies et souvent utilisées dans la literature :

$EF\varphi \equiv E(true \cup \varphi)$	« φ potentiellement »
$AF\varphi \equiv A(true \cup \varphi)$	« φ inevitablement »
$EG\varphi \equiv \neg AF\neg\varphi$	« toujours potentiellement φ »
$AG\varphi \equiv \neg EF\neg\varphi$	« invariablement φ »
$AX\varphi \equiv \neg EX\neg\varphi$	« pour tout chemin Next φ »

Exemples de formules CTL

1. EFp il est possible que p devienne vraie dans le futur, au sens où il existe une exécution du modèle pour laquelle p sera vraie (ne peut tre exprimée en LTL, Fp indique que p deviendra vraie pour toutes les exécutions) ;
2. $AG(alert \Rightarrow EFhalt)$ pour toutes les exécutions du modèle, dès lors que `alert` est vraie, il est possible d'étendre ces exécutions pour que `halt` devienne vraie.

1.7 Comparaison entre les logiques linéaires et arborescentes

Il existe plusieurs aspects à prendre en compte pour le choix d'une logique temporelle particulière. Nous présentons ci-dessous une comparaison entre les logiques temporelles linéaires et arborescentes selon différents critères.

1.7.1 Pouvoir d'expression

Un aspect important est le pouvoir d'expression d'une logique temporelle, c'est-à-dire sa capacité de décrire différentes classes de propriétés. De ce point de vue, les logiques LTL et CTL sont incomparables, chacune d'entre elles permettant de décrire des propriétés qui ne sont pas exprimables dans l'autre. Par exemple, du fait de l'absence des quantificateurs sur chemins, LTL ne permet pas de discerner l'atteignabilité potentielle de l'atteignabilité inévitable, qui s'expriment respectivement en CTL par les formules $EF\phi$ et $AF\phi$.
Inversement, du fait que les quantificateurs sur les chemins sont attachés aux modalités linéaires, CTL ne permet pas d'exprimer les propriétés nécessitant l'imbrication des modalités de chemins (notamment, certaines propriétés d'équité) : par exemple, le fait qu'une propriété est vérifiée continuellement sur chaque chemin du modèle (sauf éventuellement sur un préfixe fini de ce chemin), exprimé en LTL par la formule $FG\phi$, n'a pas de formulation équivalente en CTL.

1.7.2 Efficacité d'évaluation

En pratique, il est important de disposer d'algorithmes efficaces pour évaluer les formules temporelles sur des modèles. De ce point de vue, les logiques purement arborescentes sont avantagées par rapport aux logiques purement linéaires. En effet, les algorithmes classiques de vérification de LTL [27], basés sur la traduction des formules vers des automates de Bûchi, ont une complexité linéaire en taille du modèle et exponentielle en taille de la formule. En revanche, l'évaluation de CTL est beaucoup plus efficace : les algorithmes classiques [28] sont linéaires en taille du modèle et aussi en taille de la formule.

1.8 Conclusion

La vérification est un processus utilisé pour démontrer la correction fonctionnelle d'une conception d'un système, généralement décrit au moyen d'un langage de haut niveau (ou formel) ayant une sémantique opérationnelle bien définie, comme les algébres de processus par exemple.

Dans l'approche basée sur les modèles, très largement utilisée car caractérisée par un bon compromis coût/performance, cette description est ensuite traduite automatiquement vers un modèle sous-jacent, qui est souvent un système de transitions étiquetées (STE), c'est-à-dire un graphe (ou automate) contenant, éventuellement avec certaines abstractions, tous les comportements possibles du système. La vérification consiste alors à comparer le modèle du système avec sa spécification ou les propriétés qui décrivent le fonctionnement attendu du système (ou encore les services que celui-ci doit fournir). La spécification de propriétés consiste à décrire les propriétés du système dans un formalisme adapté pour faciliter leur vérification. Les propriétés à spécifier peuvent être de plusieurs types, mais concernent toutes des comportements que doit réaliser ou non le système. Les langages des logiques temporelles sont bien adaptés à la spécification des propriétés.
Plusieurs outils de vérification existent et utilisent diverses approches de vérification et diverses méthodes de réduction afin de pallier au problème de l'explosion combinatoire.

Dans le chapitre suivant, nous présentons les techniques permettant la génération distribuée de l'espace d'état d'un système.

Chapitre 2

Génération distribuée de l'espace d'états

2.1 Introduction

La vérification basée sur les modèles (model-checking) souffre, comme nous l'avons vu dans le chapitre précédent, du problème de l'explosion combinatoire de l'espace d'états. Parmi les techniques utilisées pour pallier à ce problème, une approche exploite les avantages du calcul parallèle et distribué. Dans cette approche, plusieurs machines coopèrent pour générer et stocker l'espace d'états, et réaliser la vérification. Dans ce cas, à la fois l'espace d'états et les charges de calcul sont répartis entre les différentes machines. Il est intéressant d'explorer les différentes techniques qui peuvent être utilisées pour accomplir la vérification dans le cas distribué ainsi que les avantages et inconvénients de cette approche. Notre étude recouvre toutes les étapes nécessaires pour une distribution de la vérification par model-checking. Les points suivants seront alors considérés :

- La génération distribuée de l'espace d'états, et la distribution ou la répartition de ce dernier.
- La vérification parallèle des propriétés du système sur l'espace d'états réparti.

Dans ce chapitre, nous nous intéressons uniquement à la génération distribuée de l'espace d'états. La vérification fera l'objet des chapitres suivants.

La génération de l'espace d'états est souvent une étape essentielle pour la vérification et l'analyse formelle des systèmes. Le but est d'obtenir l'ensemble des états atteignables à partir de l'état ou des états initiaux des systèmes décrits à l'aide de modèles de haut niveau à états discrets. L'espace d'états peut ensuite être utilisé pour répondre à des questions simples telles que « existe-t-il une état de blocage ? » ou à des questions plus complexes exprimées dans une logique temporelle.

La génération de l'espace d'états consiste en la construction du modèle de base sur lequel peut s'effectuer la vérification. Ce modèle est souvent un système de transitions, composé d'un ensemble de nœuds représentant l'ensemble des états du système, et d'un ensemble d'arcs qui représentant les transitions entre les différents états.

Les systèmes ou les programmes à vérifier sont généralement décrits en utilisant un langage ou un modèle de description de haut niveau tel que les algèbres de processus, les réseaux de Petri. L'espace des états est alors généré à partir de ces modèles afin d'effectuer vérification et analyse de ces systèmes.

2.2 Les réseaux de Petri

Le modèle des Réseaux de Petri [52, 17, 30], introduit en 1962 par Carl Adam Petri, est l'un des modèles les plus utilisés pour la description et l'analyse des systèmes concurrents. Il possède l'avantage d'avoir à la fois une représentation graphique et une sémantique formelle, permettant de représenter de manière simple et claire les concepts de parallélisme, de synchronisation, de partage de ressources ... Un grand nombre de propriétés peuvent être vérifiées sur les réseaux de Petri, par exemple l'absence de blocage, la disponibilité permanente d'une fonctionnalité, ... Pour étudier ces propriétés, de nombreuses techniques de vérification ont été développées, utilisant des outils variés tels que la théorie des graphes, l'algèbre linéaire, la théorie des langages.

Nous décrivons ce modèle dans la suite de cette section, pour nous intéresser ensuite à la génération répartie de son espace d'états ainsi qu'à la vérification distribuée des propriétés usuelles.

Un réseau de Petri peut être vu comme un graphe biparti avec deux types de nœuds, les places et les transitions, où les arcs relient soit les places aux transitions soit les transitions aux places. Les places sont représentées par des cercles ou des ellipses, et les transitions

par des barres ou des rectangles. Les arcs sont pondérés, les poids correspondant à 1 ne sont pas représentés.

Définition 8 *(Réseau de Petri) Un* réseau de Petri *(RdP) est un triplet* $\mathcal{N} = \langle P, T, W \rangle$ *où :*

- P *est un ensemble fini de* places *;*
- T *est un ensemble fini de* transitions *tel que :* $P \cap T = \emptyset$ *et* $P \cup T \neq \emptyset$ *;*
- $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ *est la fonction de* valuation *(poids).* $W(p,t)$ *(resp.* $W(t,p)$*) est le poids de l'arc reliant la place* p *à la transition* t *(resp. reliant la transition* t *à la place* p*).*

La fonction de valuation W regroupe les préconditions et les postconditions de franchissement des transitions. Celles-ci peuvent être définies séparément à l'aide des fonctions suivantes :

- **Pré** : $P \times T \longrightarrow \mathbb{N}$ est la restriction de W à $P \times T$ exprimant les préconditions requises pour qu'une transition soit franchissable. $Pré(p,t) = W(p,t)$.
- **Post** : $P \times T \longrightarrow \mathbb{N}$ est la restriction de W à $T \times P$ exprimant les conséquences du tir d'une transition. $Post(p,t) = W(t,p)$.

Pour définir l'état d'un système modélisé par un réseau de Petri, il est nécessaire de compléter le réseau de Petri par un marquage. Ce marquage est un nombre entier (positif ou nul) de marques ou jetons (représentés par des points noirs) dans chaque place du réseau de Petri. Le nombre de jetons contenus pour un marquage M dans une place p_i sera noté $M(p_i)$. Le marquage du réseau peut alors être défini par le vecteur M. Le marquage initial définit l'état initial du système.

Définition 9 *(Réseau de Petri marqué) Un* réseau de Petri marqué $\langle \mathcal{N}, M_0 \rangle$ *est un réseau de Petri* $\mathcal{N}$ *avec un marquage initial* $M_0 : P \longrightarrow \mathbb{N}$*. Le marquage initial de la place* $p \in P$ *est* $M_0(p)$*.*

Le comportement dynamique des systèmes à événements discrets est donné à l'aide de leurs états et leurs évolutions. Les marquages et les transitions permettent de représenter un tel comportement. L'évolution obéit aux règles de franchissement suivantes :

Définition 10 *(Transition franchissable) Une transition $t \in T$ est dite* franchissable *pour le marquage M si et seulement si*

$$\forall p \in P : M(p) \geqslant W(p,t)$$

Ceci est noté $M[t\rangle$.

Définition 11 *(Franchissement d'une transition) Une transition $t \in T$ franchissable pour le marquage M peut être franchie ou tirée, conduisant à un nouveau marquage M' tel que :*

$$\forall p \in P : M'(p) = M(p) - W(p,t) + W(t,p)$$

Ceci est noté $M[t\rangle M'$.

Le franchissement peut être étendu aux séquences de transitions. Ainsi, une séquence $\sigma = t_1 t_2 \dots t_n$ est franchissable à partir d'un marquage M ssi :

$$\exists M_1, M_2, \dots M_n \text{ tels que } M[t_1\rangle M_1[t_2\rangle M_2 \dots [t_n\rangle M_n$$

Le franchissement de la séquence σ est est noté $M[\sigma\rangle M_n$.

Si un marquage M_n peut être obtenu à partir d'un marquage M_0 par le tir d'une séquence de transitions, M_n est alors dit *atteignable* ou *accessible*.

L'ensemble des marquages atteignables à partir d'un marquage M est noté $M[\,\rangle$.

On peut construire le graphe des marquages accessibles, qui est un graphe orienté dont les sommets correspondent aux marquages accessibles à partir du marquage initial M_0, et où un arc étiqueté par la transition t relie deux sommets associés aux marquages M_1 et M_2, si $M_1[t\rangle M_2$.

Définition 12 *(Graphe des Marquages accessibles) Soit $\langle \mathcal{N}, M_0 \rangle$ un réseau de Petri marqué. Le graphe des marquages accessibles $G(\mathcal{N}, M_0)$ est défini par :*

- *L'ensemble des sommets : chaque sommet représente un marquage accessible.*
- *L'ensemble des arcs : il existe un arc étiqueté par t de M à M' ssi $M[t\rangle M'$.*

Etant donné un modèle de spécifcation décrivant un système, la première étape à effectuer dans le cadre d'une vérification par le model-checking est la construction de son espace d'états. L'approche de construction consiste à explorer les différents états que l'on

peut atteindre à partir de l'état initial. Cette étape est appelée génération de l'espace d'états.

L'algorithme 1 décrit la construction du graphe des marquages accessibles pour un réseau de Petri marqué $\langle \mathcal{N}, M_0 \rangle$. Il utilise un ensemble *Visited* de sommets visités mais non encore explorés. Un sommet est dit exploré si tous ses successeurs ont déjà été construits.

Procedure 1 Reachability Graph $G(\mathcal{N}, M_0)$

Input : un réseau de Petri marqué $\langle \mathcal{N}, M_0 \rangle$
Output: son graphe d'accessibilité $G = (S, A)$

```
begin
    Visited ← M0
    S ← M0
    while Visited ≠ ∅ do
        Choose M ∈ Waiting
        forall the t ∈ T tq M[t⟩M' do
            if M' ∉ S then
                Visited ← Visited ∪ {M'}
                S ← S ∪ {M'}
            A ← A ∪ {(M, t, M')}
        Visited ← Visited \ {M}
end
```

2.3 Génération distribuée de l'espace d'états

La distribution de la génération de l'espace d'états est un problème qui a été largement étudié ces dernières années. On peut trouver différentes approches et plusieurs outils de génération et d'exploration distribuée de l'espace d'états sur un réseau (cluster) de stations [2, 10, 19, 31, 43].

Plusieurs approches peuvent être envisagées afin d'effectuer la génération distribuée de l'espace d'états. Parmi ces approches on trouve :

– Celle où l'on considère deux types de processus : un processus ***générateur*** qui calcule

les états et les transmet à des processus de ***stockage*** en utilisant une fonction de hashage h. L'ensemble *Visited* est géré par le processus générateur.

- Une autre approche utilise un processus ***coordonnateur*** et des processus ***explorateurs***. Le processus coordonnateur initie la génération en envoyant l'état initial s_0 au processus $h(s_0)$. C'est aussi lui qui se charge de la détection de la terminaison. Chacun des processus explorateurs génère et stocke une partie distincte de l'espace d'états. Chacun a donc son propre ensemble *Visited*.
- Dans une approche similaire, un des processus, désigné par élection, joue le rôle du processus coordonnateur.

Nous avons opté pour la deuxième approche, qui permet de tirer un meilleur avantage du cluster pour effectuer une génération de l'espace d'états avec un degré de parallélisation plus intéressant.

Dans ce qui suit, nous confondrons sciemment *marquages* et *états*.

La génération distribuée consiste donc à partitionner l'ensemble des états sur un ensemble de N stations connectées en réseau et communiquant par échange de messages. Ce partitionnement s'effectue grâce à une fonction de répartition (fonction de hashage) $h : S \longrightarrow P$ qui associe à chaque état s un entier $h(s)$ correspondant au numéro du processus propriétaire de s. S est donc l'ensemble des états et $P = \{p_1, p_2, \ldots, p_N\}$ est l'ensemble des processus (ou stations) participant à la génération de l'espace d'états.

Ainsi, chaque processus p_i construit et sauvegarde une partie de l'espace d'états (S_i, A_i), comme l'illustre la figure 2.1.

- $S_i = \{s \in S / h(s) = i\}$ tels que $\bigcup_{i=0}^{N-1} = S$ et $\forall i \neq j, S_i \cap S_j = \emptyset$ (l'ensemble des états est partitionné entre N machines).
- $A_i = \{(s, s') / s \in S_i\}$, représentent l'ensemble des arcs ayant leur source dans S_i.

L'arc (s, s') sera dit **arc interne** au processus i si s est s' appartiennent à S_i. Il sera dit **arc traversant** si $s \in S_i$ et $s' \notin S_i$. Le sommet s sera dit alors **sommet local** et s' **sommet distant**.

L'algorithme 2 décrit les étapes exécutées par le processus i pour la génération de l'espace d'états local (S_i, A_i).

L'algorithme de génération effectue les opérations suivantes :

- choisit un sommet s de l'ensemble $visited_i$, génère ses successeurs.

Procedure 2 DistributedGeneration()

```
begin
    NotEnd ← true
    while NotEnd do
        while Visited_i ≠ ∅ do
            Choose M ∈ Visited_i
            forall the t ∈ T such that M[t⟩M' do
                if h(M') ≠ i then
                    /* Le nouvel état M' n'appartient pas à ce processus
                       */
                    Send (h(M'), M')
                else
                    /* M' appartient au processus i */
                    if M' ∉ S_i then
                        Visited_i ← Visited_i ∪ {M'}
                        S_i ← S_i ∪ {M'}
                A_i ← A_i ∪ {(M, t, M')}
            Visited_i ← Visited_i \ {M}
            if Visited = ∅ then
                Send (coordinator, Inactif)
end
```

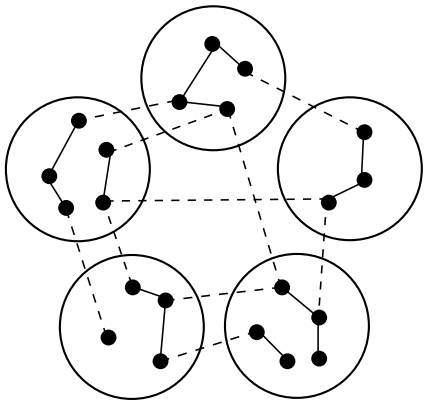

FIGURE 2.1 – Représentation d'un espace d'états réparti sur plusieurs machines.

- chaque successeur s' obtenu par tir de la transition t :
  - est inséré dans le graphe s'il est local, c'est-à-dire $h(s') = i$, puis ajouté à l'ensemble $Visited_i$ s'il est nouveau.
  - est envoyé au processus $h(s')$ si $h(s') \neq i$.
- insère ensuite l'arc (s, t, s').

Les communications sont asynchrones, le traitement des messages s'effectue par préemption. Ainsi, à la réception d'un message, le processus récepteur est interrompu et une procédure de traitement de messages *MessHandler*() est alors exécutée. Une portion de cette procédure est donné dans l'algorithme 15, qui décrit les opérations exécutées lorsqu'un processus reçoit un nouvel état s, ou lorsqu'il reçoit un message de terminaison.

Chaque message contient entre autres un champ qui indique le type du message et un champ de donnée qui peut contenir, par exemple, l'état transmis.

Parmi les messages échangés, il y a évidemment ceux permettant la détection de la terminaison. Celle-ci se produit lorsque :

- Les ensembles de sommets *Visited* de tous les processus sont vides.
- Aucun message n'est en transit, c'est-à-dire que tous les messages envoyés sont reçus par leur destinataire.

Le principe utilisé pour la détection de la terminaison est simple et peut être décrit comme suit :

- Lorsque l'ensemble des sommets visités, *Visited*, d'un processus $explorateur_i$ est vide,

Procedure 3 MessageHandler()

```
begin
    ⋮
    if Message.Type = STATE then
        /* Le message contient un état à traiter */
        Visited_i ← Visited_i ∪ {Message.State}
    if Message.Type = TERMINATE then
        /* Le message de terminaison d'un processus est reçu */
        NotEnd ← false
    ⋮
end
```

il envoie un message qu'il est dans l'état **oisif** au processus *coordonateur* et se met en *d'attente*.

- A la réception d'un tel message, le processus *coordonateur* retient que le processus $explorateur_i$ est dans l'état *Inactif*.
- Si un processus $explorateur_j$, qui est dans l'état *Inactif*, reçoit un nouveau marquage à explorer, il l'ignore si celui-ci existe déjà dans son espace local. Sinon, il le met dans sa liste de sommets à explorer $Visited_j$ puis envoie un message de changement d'état au coordonnateur. Ce dernier, met à jour l'état du processus $explorateur_j$ à *Actif*.
- Chaque processus $explorateur_i$ envoie avec le message de terminaison son identifiant, le nombre de messages émis et le nombre de messages reçus. Le coordonateur vérifie, d'une part si tous les processus sont dans l'état *Inactif*, d'autre part que le nombre total de messages émis par l'ensemble des processus est égal au nombre total de messages reçus. Il envoie alors un message de **terminaison** en diffusion pour l'ensemble des processus.

Cette procédure de détection de la terminaison est différente de celle utilisée dans la version parallèle de Spin [6] qui est complètement centralisée. L'avantage de cette procédure est qu'elle permet de réduire le nombre de messages échangés entre les différentes stations.

2.4 Performances des algorithmes de génération distribués

Les performances des algorithmes de génération et de vérification basés sur l'espace d'états distribué dépendent fortement de l'équilibre de charge des différents processus et du coût de communication, qui correspond au nombre de messages échangés entre les différents processus.

Le choix de la fonction de répartition joue un rôle prépondérant. Il faut donc construire une fonction qui permet de garantir un équilibre de charge, c'est-à-dire l'affectation de parties d'espaces d'états plus ou moins égales en terme de nombre d'états, entre les différentes machines. Il faut aussi qu'elle permette de minimiser le nombre d'arcs traversants afin de réduire le nombre de messages échangés.

Il est à noter que le choix de la fonction de répartition n'est pas facile, vu l'irrégularité des données manipulées, et dépend fortement du modèle à analyser, du formalisme de modélisation et de la puissance des machines utilisées. Donc, selon ces critères, la fonction de répartition doit être choisie avec une étude préalable de l'évolution du système modélisé.

2.4.1 Parallélisme et équilibrage de charge

L'équilibre de charges dans les système distribués peut être analysé selon plusieurs aspects : l'équilibre spatial, la localité, et l'équilibre temporel.

L'équilibre spatial

L'espace mémoire demandé pour stocker le sous ensemble d'états S_i doit être environ le même par chaque processus i. Cela est fondamental, puisque la mémoire est la ressource principale qui cause le goulet d'étranglement dans la génération de l'espace d'états. L'équilibre spatial peut être mesuré comme suit :

$$\max \frac{S_i}{S_j} \qquad \forall i, j \in \{1 \dots N\}$$

Une bonne répartition doit avoir un équilibre spatial proche de 1.

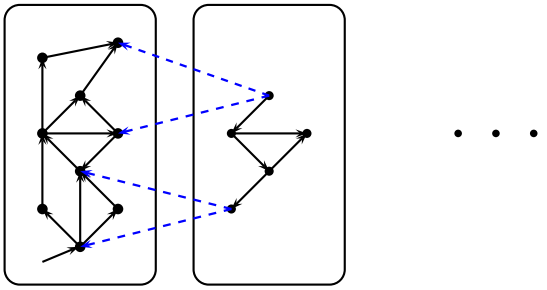

FIGURE 2.2 – Charge non équilibrée

La localité

Le nombre d'arcs reliant des états appartenant à deux propriétaires différents, appelés arcs traversants (cross arcs), doit être aussi réduit que possible. Un grand nombre d'arcs traversants engendre un nombre important d'échanges de messages et réduit ainsi les performances de l'algorithme distribué.

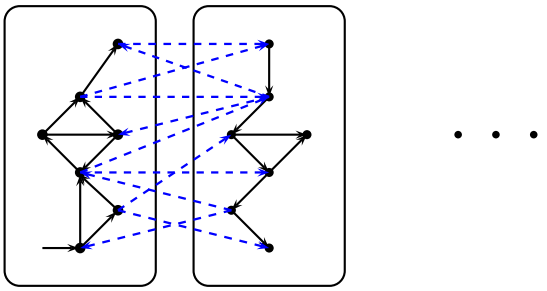

FIGURE 2.3 – Trop d'arcs traversants

Cependant, même si l'on assure une bonne localité, c'est-à-dire un nombre restreint d'arcs traversants, et un bon équilibre spatial, cela n'implique pas forcément que l'algorithme distribué ait de bonnes performances.

L'équilibre temporel

La figure 2.4 illustre bien que même si l'équilibre spatial et la localité sont assurés il n'en demeure pas moins que pour garantir de bonnes performances des algorithmes distribués,

il faut aussi assurer que les processus soient le moins possible inactifs.

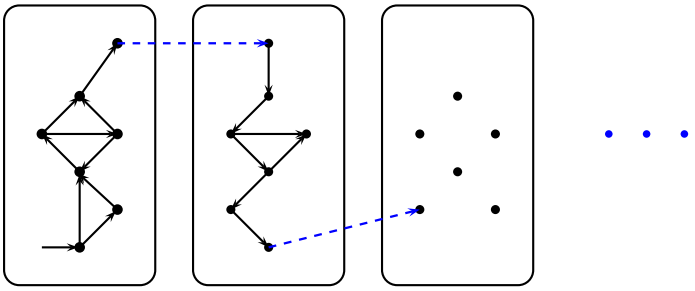

FIGURE 2.4 – Mauvais équilibre temporel

Dans cet exemple extrême, un seul arc relie le sous-ensemble d'états S_i au sous-ensemble S_{i+1}. Dans ce cas seul un processus peut être actif à la fois, et tous les autres sont inactifs. Ainsi, le traitement est effectué de manière quasi-séquentielle.

Pour obtenir de bonnes performances des algorithmes distribués, on doit garantir une charge de calcul équilibrée entre les processus et le temps d'inactivité doit être réduit au minimum possible.

Mais la construction d'une fonction de hashage qui assure à la fois un équilibre spatial et temporel reste très difficile.

2.4.2 Choix de la fonction de répartition

Il y a deux types de fonctions de répartition : des fonctions dites statiques ou des fonctions dynamiques.

Fonction statique

Dans ce cas, la fonction de répartition est généralement définie comme suit :

$$hash(i) = [\rho M(p_1) + \rho^2 M(p_2) + \ldots + \rho^k M(p_k)] \mod N$$

Où ρ est un nombre premier, $M(p_1), M(p_2), \ldots, M(p_k)$ correspondent, respectivement, aux marquages de k places $p_1, p_2, \ldots, p_k$ choisies.

Ces paramètres sont en général fixés par l'utilisateur, selon la nature du réseau de Petri et la configuration du graphe des marquages ainsi que des propriétés qu'il souhaite analyser. La sélection du sous-ensemble de places sur lesquelles est basée la fonction de hashage peut influer sur la qualité de la répartition.

L'avantage de donner à l'utilisateur la possibilité de définir la fonction de hashage permet à celui-ci de spécifier ses buts, par exemple, s'il choisit $\rho = 1$ et considère que les places ou le marquage peut uniquement diminuer ou augmenter de 1 à chaque franchissement, la communication entre processus se réalise seulement entre les processus d'indices contigus. Cette propriété est très importante lorsque l'interconnexion entre les processus se fait à travers un réseau en anneau.

De plus, le franchissement de transitions non reliées aux places choisies, conduit à des marquages appartenant au même processus, minimisant ainsi le nombre d'arcs traversants.

Par contre, un mauvais choix peut présenter quelques inconvénients : par exemple choisir k places constituant un invariant de la forme : $M(p_1) + M(p_2) + \ldots + M(p_k) = c$, donne comme processus propriétaire de tous les marquages le processus c. Par conséquent tous les autres processus sont inactifs. De plus, on peut avoir une plus grande partie de l'espace d'états concentrée sur certains processus seulement, créant ainsi un déséquilibre spatial de charge.

Fonction dynamique

La définition d'une fonction de répartition statique demande l'assistance de l'utilisateur qui fournit des paramètres nécéssaires au calcul de cette fonction. Ces paramètres sont souvent très difficiles à choisir, et s'ils ne sont pas judicieusement choisis, ils peuvent conduire à un déséquilibre spatial. Une autre approche a été proposée dans laquelle la fonction de répartition peut être ajustée de manière dynamique [20]. Ainsi, on définit une application intermédiaire qui associe à chaque état S la classe C à laquelle il appartient. Le nombre de classes étant suffisamment grand (égal, par exemple à $100 \times N$, N étant le nombre de processus), Ces classes sont affectées de manière dynamique aux différents processus afin de préserver l'équilibre de charge.

$Class : S \to \{1, \ldots, C\}$

$Proc : \{1, \ldots, C\} \to \{1, \ldots, N\}$

La génération de l'espace d'états est interrompue périodiquement pour vérifier si l'équili-

bre de charge est assuré, en comparant par exemple la taille de S_i pour chacun des processus i. Dans le cas où l'on détecte un déséquilibre de charge, on peut procéder à une autre réaffectation des classes aux processus.

2.4.3 L'équilibre de charge par la virtualisation de processeurs

Le concept de virtualisation de processeurs a été présenté pour la première fois dans [42]. L'idée de base est de distribuer le calcul sur un ensemble de processeurs virtuels, dont le nombre doit être plus grand que le nombre de processeurs physiques. Le système se charge d'affecter les processeurs virtuels aux processeurs physiques. Ainsi, chaque processeur physique peut accueillir un ou plusieurs processeurs virtuels.

La virtualisation de processeurs offre une véritable flexibilité. Elle permet, par exemple, une décomposition naturelle du problème à traiter ou de considérer une topologie qui serait impossible à réaliser en raison des limites physiques de l'environnement de calcul dans lequel on se trouve.

La virtualisation de processeurs possède plusieurs avantages [41], parmi lesquels :

- une meilleure utilisation des processeurs ;
- un équilibre de charge plus facile à réaliser.

En effet, même pendant l'exécution, lorsque l'on traite un problème avec des données irrégulières et dont l'évolution est imprévisible, le mécanisme de migration de processeurs virtuels d'un processeur physique à un autre constitue une solution adéquate pour rééquilibrer leurs charges.

La génération de l'espace d'états et les algorithmes de model-checking sont aussi des applications considérées comme irrégulières [44]. L'utilisation de la virtualisation, grâce aux outils offerts par le système d'exécution de Charm++ pris en charge dans AMPI (Adaptive Message Passing Interface), permet alors de pallier à ce problème.

Dans [56], nous avons proposé une solution, basée sur la virtualisation de processeurs, qui implique une large répartition de l'espace d'états, sur des processeurs virtuels, dont le nombre est bien plus grand que le nombre de processeurs physiques. Cette répartition est effectuée de manière dynamique. En effet, charm++ peut affecter un nouveau processeurs virtuel automatiquement au processeurs physique le moins chargé. La création d'un nouveau processeurs virtuel se fait lorsqu'on génère le premier état appartenant à ce dernier. Des processeurs virtuels peuvent aussi migrer d'un processeur physique dont la charge est importante à un autre moins chargé de sorte à assurer l'équilibre.

Comme nous l'avons souligner auparavant, il est difficile de construire une fonction de répartition qui assure à la fois un équilibre spatial est temporel. L'équilibrage de charge étant assuré par charm++, cela permet donc de consacrer tout les efforts à la construction d'un fonction de répartition réduisant le nombre d'arcs traversant par exemple.

Les algorithmes utilisés sont les mêmes que ceux proposés précédemment. Chaque processeur virtuel se charge de générer un fragment de l'espace d'états.

2.4.4 Utilisation d'un cache mémoire

Une autre technique permettant d'améliorer les performances de l'algorithme est l'utilisation d'un cache mémoire. En effet, chaque processus envoie à leurs proprétaires les états générés qui ne lui appartiennent pas. Or un état peut être atteint par plusieurs chemins différents, cette dernière situation implique donc l'envoi d'un même état plusieurs fois, ce qui engendre des messages inutiles diminuant seulement les performances de l'algorithme de génération. Pour pallier à ce problème, nous réservons sur chaque machine un espace mémoire, fonctionnant comme un cache d'états, dans lequel sont enregistrés les derniers états ayant été envoyés aux autres processus. Chaque processus i vérifie d'abord pour tout état généré s tel que $h(s) \neq i$, s'il n'existe pas dans son cache, et c'est uniquement dans ce cas que l'état s sera transmis au processus $h(s)$, réduisant ainsi le nombre de messages échangés.

2.5 Tests

Les algorithmes présentés ont été implémentés et testés sur un prototype pouvant s'exécuter sur plusieurs stations liées en réseaux et communiquant par l'envoi de messages. Ces tests ont été effectués sur une plateforme constituée de 12 stations fonctionnant sous le système d'exploitation Linux, et ayant chacune 512 MO de RAM et un processeur Pentium IV. Ces stations sont reliées par un réseau Ethernet. Le prototype réalisé permet l'analyse distribuée des propriétés qualitatives classiques des réseaux de Petri, telles que la vivacité et les états d'accueil. Des propriétés spécifiques exprimées en logique temporelle CTL peuvent également être analysées. Nous donnons dans ce chapitre uniquement les résultats des tests concernant la génération de l'espace d'états. Pour nos tests, nous avons consideré le problème classique des philosophes et le problème de gestionnaires de bases de données distribuées.

2.5.1 Problème des philosophes

Le problèmes des philosophes a été introduit par Dijkstra pour illustrer le problème d'allocation de ressources entre plusieurs processus. Il comprend un certain nombre de philosophes assis autour d'une table circulaire. Il y a devant chaque philosophe un plat et entre deux philosophes une fourchette. Chaque philosophe passe son temps soit à penser soit à manger. Il peut arbitrairement décider de manger ou d'arrêter de manger et de se mettre à penser. Lorsqu'il décide de manger, il a besoin des deux fourchettes, celle à sa gauche et celle à sa droite.

Les résultats obtenus sont donnés dans le tableau 2.1. La dernière colonne indique le temps de génération dans le cas séquentiel. Un état dans lequel i philosophes sont en train de manger n'est relié qu'aux états où il y a $i-1$ ou $i+1$ philosophes en train de manger. Ainsi, nous avons construit une fonction de répartition qui permet de stocker le maximum d'états voisins sur une même station.

Nb Phil	Nb States	Nb Trans (N_1)	Nb Cross. arcs (N_2)	N_1/N_2	Nb Mess. (N_3)	N_3/N_2	Time G.(sec)	1 proc Time
5	11	30	18	1.67	36	2.00	<0.01	<0.01
10	123	680	242	2.81	494	2.03	<0.01	0.01
15	1,364	11,310	2,731	4.14	5477	2.01	0.06	0.38
20	15,127	167,240	30,236	5.53	60,493	2.00	2.52	13.48
25	167,761	2,318,400	315,718	7.34	631,587	2.00	32.58	216.09
30	1,728,813	28,686,031	3,572,821	8.03	7,156,116	2.00	7547.45	$\gg$[1]

TABLE 2.1 – Distributed generation

On peut noter de ce tableau, que le nombre d'arcs traversants ne représente que 1/8 du total des arcs. Ce qui peut être consideré tout à fait raisonnable au vu de la densité du graphe des marquages accessibles. Le nombre de messages est proportionnel au nombre de transitions traversantes, environ 2 messages pour chaque transition traversante, comme cela est indiqué par le ratio N3/N2.

2.5.2 Problème des gestionnaires de base de données distribuée

Le problème des gestionnaires de base de de données distribuée consiste en N sites différents, chacun contenant une copie de la base de données. Cette copie est gérée par

1. Ce problème, vu sa taille, n'a pas pu être traité sur une seule machine.

une gestionnaire (manager) de bases de données local. Chaque gestionnaire est autorisé à effectuer des mises à jour dans sa propre copie, mais doit informer tous les autres gestionnaires afin qu'ils opèrent la même mise à jour sur leurs copies.
Un gestionnaire de base de données peut être dans l'un des trois états : *Inactive*, *Waiting* (en attente d'un accusé) ou *Performing* (effectuant une mise à jour).
Lorsqu'un gestionnaire k décide d'effectuer une mise à jour, il envoie à tous les autres gestionnaires un message, pour qu'ils puissent effectuer à leur niveau la même mise à jour, se met dans l'état *Waiting* et attend de recevoir les confirmations de mise à jour.
Lorsqu'un gestionnaire $l \neq k$ reçoit un message de la part du gestionnaire k pour effectuer une mise à jour, il passe de l'état *Inactive* à l'état *Performing*, effectue la mise à jour, envoie un message de confirmation *Acknowledgment* au gestionnaire k pour l'informer que la mise à jour a été effectuée, puis se remet dans l'état *Inactive*.
Lorsque le gestionnaire k (initiateur de la mise à jour) a reçu toutes les confirmations, il se remet dans l'état *Inactive*. Après quoi, les autres gestionnaires peuvent initier des mises à jour.

Ce modèle permet de trouver facilement une bonne fonction de répartition : par exemple, choisir de regrouper tous les états dans lesquels le gestionnaire i a initié une mise à jour dans la station $i \mod n$.
Les résultats des tests relatifs au problème des gestionnaires de base de données distribuée sont donnés dans le tableau 2.2.

Nb Managers	Nb States	Nb Trans.	Nb Trav. Trans.	Nb Mess.	Time (s)	1 proc Time
6	1,460	4,878	14	208	<0.01	<0.01
7	5,105	20,433	15	224	0.27	0.01
8	17,498	81,688	16	245	1.33	0.12
9	59,051	314,955	19	282	2.68	3.85
10	196,832	1,181,010	22	293	131.73	521.5

TABLE 2.2 – Problème des gestionnaire de base de données distribuée

Malgré l'augmentation de la taille de l'espace d'états, le nombre de messages n'augmente que légèrement, et ce grâce au choix d'une fonction de répartition adéquate, qui a permis d'avoir un nombre réduit d'arcs traversants.

2.5.3 Equilibrage de charge par la Virtualisation de processeurs

Pour montrer l'impact de l'utilisation de la virtualisation de processeurs sur l'équilibre de charges des différents processus, nous avons aussi effectué des tests avec un problème de 24 philosophes géneérant un nombre total de 103682 états, réparti sur les 06 machines. Les figures 2.5

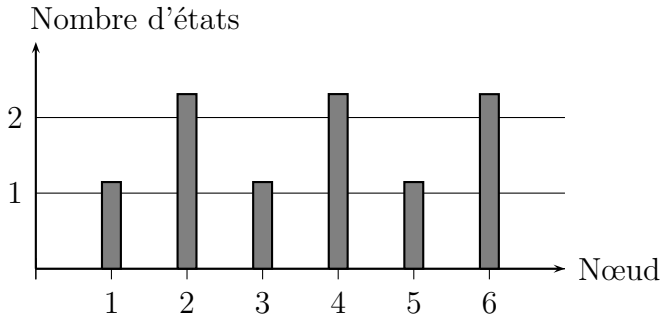

FIGURE 2.5 – Répartition des états avec une fonction de hashage statique

La répartition est bien équilibrée lorsqu'on utilise la virtualisation comme le montre la figure 2.6. Ces résultats sont obtenus avec 200 processeurs virtuels.

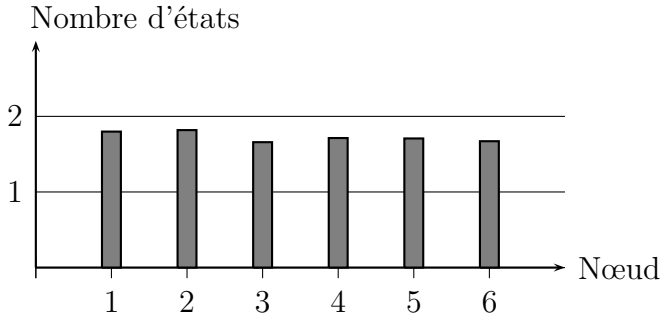

FIGURE 2.6 – Répartition des états avec la virtualisation de processeurs

2.6 Conclusion

L'utilisation d'un cluster pour la génération distribuée de l'espace d'états peut permettre de pendre en charge des systèmes de grande taille. Ainsi, selon le nombre de machines et leurs capacités, des systèmes de tailles plus ou moins importantes peuvent donc être vérifiés.

Les performances de l'algorithme de génération distribuée de l'espace d'états sont fortement liées à la fonction d'attribution des états aux machines. Le formalisme utilisé et la manière dont le système évolue sont les points clés pour choisir une bonne fonction d'attribution. Ces performances peuvent être améliorées par l'utilisation d'un cache mémoire. Ce dernier est également un moyen de minimiser les échanges de messages entre machines.

Aussi, l'utilisation de mécanismes pour assurer un équilibre de charge permet une amélioration significative des performances de ces algorithmes.
Les tests menés confirment que si l'on choisit une fonction de répartition adéquate, on peut réduire considérablement le nombre d'arcs traversants et par conséquent le nombre de messages échangés.

Même si le gain en terme de taille des problème traités reste encore limité, il n'en demeure pas moins que l'accélération *super-linéaire* pour la génération de l'espace d'états obtenue est tout de mêême un résultat appréciable.

Chapitre 3

Vérification distribuée des propriétés

3.1 Introduction

Dans ce chapitre, nous nous intéressons à la vérification distribuée de propriétés générales de réseaux de Petri : l'atteignabilité, le blocage, la vivacité et l'existence d'un état d'accueil. Certaines de ces propriétés nécessitent seulement l'exploration de l'espace d'états, qui peut être effectuée à la volée. D'autres, par contre, nécessitent d'avoir en mémoire non seulement tout l'espace d'états du système à vérifier mais aussi les transitions entre les sommets.

Ces propriétés sont connues pour être décidables pour les réseaux de Petri bornés. Néanmoins l'espace et le temps nécessaires à leur vérification sont généralement exponentiels.

Il existe aussi d'autres propriétés dont la vérification s'effectue par simple exploration de l'espace d'états et que l'on peut également mener en parallèle, par exemple le caractère borné.

Les algorithmes de vérification distribués présentés dans ce chapitre sont tous des algorithmes de parcours [64].

3.2 Vérification de propriétés d'atteignabilité

La vérification de propriétés d'atteignabilité vise à prouver qu'un état donné appartient à l'espace d'états du système. Souvent, il s'agit de vérifier si un état indésirable, correspondant à une erreur, n'est pas atteignable ou n'appartient pas à l'espace d'états (par exemple un ascenseur en mouvement alors que sa porte est ouverte).

Un état s est atteignable s'il existe un chemin partant de l'état initial s_0 et qui rencontre l'état s.

Déterminer si un état s appartient à l'espace des états peut être mené en parallèle. On continue à utiliser les deux types de processus, les processus *explorateurs* et le processus *coordonnateur*.

La recherche de l'état s peut se faire soit dans un espace d'états entièrement généré en mémoire, soit à la volée. Dans les deux cas, c'est uniquement le processus $i = h(s)$ qui vérifie si s appartient à son espace local S_i. Dans le cas où il le trouve, il envoie un message `Found` au processus *coordonnateur*.

Ceci peut être généralisé pour rechercher si un état ou un sous-ensemble d'états possédant une certaine propriété φ sur le marquage sont atteignables. Ainsi, tous les processus *explorateurs* ayant des états satisfaisant φ dans leurs états locaux renvoient `Found` avec les états satisfaisants la propriété au processus *coordonnateur*, sinon ils envoient `NotFound`. Ainsi, le processus *coordonnateur* affiche les états vérifiant φ qu'il aurait reçu avec les messages `Found`, ou conclu qu'il n'en existe pas s'il ne reçoit que des messages `NotFound`.

Si les états sont équitablement répartis sur les processus explorateurs, on peut alors espérer une accélération linéaire.

3.3 Absence de blocage

Une propriété importante que l'on souhaite souvent assurer dans un système est l'absence de blocage. Il est nécessaire de distinguer les états bloquants (blocages non souhaités) des états de terminaison (états finaux).

Un état bloquant (*deadlock*) est un état à partir duquel le système ne peut plus évoluer. Par conséquent, il ne possède aucun successeur.

Définition 13 *(État bloquant) Soit un réseau de Petri marqué $\langle \mathcal{N}, M_0 \rangle$. Un marquage accessible M est dit* bloquant *si et seulement si : $\forall t \in T : \neg M[t\rangle$.*

Un réseau de Petri est donc sans blocage si et seulement si pour tout marquage accessible M, il existe au moins une transition franchissable à partir de M : $\forall M \exists t \in T / M[t\rangle$.

La vérification de cette propriété consiste simplement à vérifier si l'espace d'états comporte un état sans successeur. Pour vérifier en parallèle cette propriété, chaque processus *explorateur* recherche dans ses états locaux si un tel état existe. Comme dans l'algorithme d'atteignabilité, les processus *explorateurs* envoient le résultat de leur recherche au processus *coordonnateur*, qui affiche alors le résultat global.

Dans les deux cas précédents cette recherche peut être menée à la volée, c'est-à-dire au fur et mesure de la génération de l'espace d'états. La génération peut alors être interrompue dès qu'on atteint l'état recherché ou lorsqu'on rencontre un état puits, même si l'espace d'états n'est pas entièrement généré.

Une accélération linéaire peut aussi être obtenue si l'espace d'états est réparti de manière équilibrée.

Il pourrait être intéressant de donner à la fin, les exécutions (chemins) ayant abouti aux cas d'erreurs ou au blocage, mais pour obtenir ces chemins, il ne suffit pas de faire un examen des états, il faut aussi effectuer des parcours arrières jusqu'à atteindre l'état initial. Ce type de parcours est similaire au parcours en profondeur d'abord qui n'est pas bien adaptée au calcul distribué. Nous avons donc choisi de fournir uniquement le plus court chemin ayant menés à ces états, et pour obtenir le plus court chemin menant de l'état initial s_0 à un état s, on définit la fonction de distance *dist* qui associe à chaque état s, $dist(s)$ telle que :

- $dist(s_0) = 0$
- $dist(s) = 1 + \min\limits_{s_j \in \Gamma^-(s)} \{dist(s_j)\}$,

où $\Gamma^-(s)$ est l'ensemble des prédécesseurs de s.

Cette fonction renvoie le nombre minimal d'arcs qu'il faut utiliser pour aller de s_0 à s.

On doit aussi conserver pour chaque état s son prédécesseur sur ce chemin, qu'on va noter $Opt(s)$, telle que :

$$Opt(s) = s' \text{ si et seulement si } s' \in \Gamma^-(s) \text{ et } dist(s') = \min_{s_j \in \Gamma^-(s)} \{dist(s_j)\}$$

La distance et le prédécesseur dans le plus court chemin de chaque sommet sont calculés pendant la génération de l'espace d'états.

Pour afficher ce chemin, qui est morcelé dans les espaces d'états locaux des différents processus *explorateurs*, il faut qu'il soit transmis au *coordonnateur*. Le processus ayant l'état final commence par envoyer cet état avec ses prédécesseurs dans le chemin. Pour cela il doit faire un parcours arrière à partir du sommet final, puis son prédécesseur et ainsi de suite, jusqu'á ce qu'il arrive sur un état prédécesseur s qui ne fait pas partie de ses états locaux. Il envoie alors un message contenant les états parcourus au *coordonnateur* et un autre message au processus $h(s)$ pour reprendre le parcours arrière sur le chemin à partir du sommet s et envoyer à son tour la partie de chemin parcourue au processus *coordonnateur*. Un processus *explorateur* peut être amené à reprendre le parcours arrière plusieurs fois. Le parcours s'arrête lorsque le processus qui détient l'état initial s_0 le rencontre lors de son parcours arrière.

On peut remarquer que pour exhiber un tel chemin, les processus doivent effectuer un parcours arrière, mais un seul processus peut effectuer ce parcours à la fois. Il n'y a donc pas de parallélisme possible pendant cette opération.

Remarque 1

Le cache introduit dans le chapitre précédent, pour réduire le nombre de messages echangés lors de la génération, doit être géré autrement. En effet, on doit retransmettre même les états ayant déjà été envoyés si la nouvelle distance est inférieure à la distance précédente. On doit donc avoir avec chaque état se trouvant dans le cache sa distance courante.

3.4 La vivacité

Un système est dit vivant s'il garde toujours la possibilité d'éffectuer toutes ses actions dans le futur. C'est-à-dire, qu'aucune action ne sera à aucun moment définitivement impossible.

Dans les réseaux de Petri, une transition est dite vivante si elle reste toujours potentiellement franchissable dans le futur.

Définition 14 *(Vivacité) Soit $t \in T$ une transition, t est* vivante *si et seulement si :*

$$\forall M \in [M_0\rangle : \exists M' \in [M\rangle s.t. M'[t\rangle$$

Un réseau de Petri $\langle \mathcal{N}, M_0\rangle$ est vivant si et seulement si toutes ses transitions sont vivantes.

En d'autres termes, quel que soit le marquage accessible atteint à partir du marquage initial M_0, il est toujours possible de franchir n'importe quelle transition $t \in T$ dans le futur. Ceci garantit qu'aucune action ne devient définitivement inaccessible.

Sur le plan pratique, la vérification qu'une transition t est vivante consiste à vérifier si elle appartient à toutes les composantes fortement connexes terminales du graphe des marquages accessibles.

Une composante fortement connexe est dite *terminale*, si aucun sommet de cette composante ne possède de successeur dans une autre composante fortement connexe.

La détermination des composantes fortement connexes et en particulier des composantes fortement connexes terminales est alors nécessaire pour la vérification d'une telle propriété, c'est ce que nous allons détailler dans la suite de ce chapitre dans le cadre distribué.

3.5 État d'accueil

Un marquage est dit état d'accueil, si et seulement si il est toujours atteignable dans le futur.

Définition 15 *(État d'accueil) Étant donné un réseau de Petri marqué $R = \langle \mathcal{N}, M_0\rangle$, un marquage M est un* état d'accueil *dans R si et seulement si :*

$$\forall M' \in [M_0\rangle : M \in [M'\rangle$$

La vérification qu'un marquage donné est un état d'accueil nécessite également de déterminer les composantes fortement connexes terminales du graphe des marquages accessibles. En effet, il faut que ce marquage appartienne à une composante fortement connexe terminale, et que celle-ci soit 'unique composante fortement connexe terminale.

3.6 Algorithme distribué de calcul des composantes fortement connexes

Parmi les algorithmes permettant de déterminer les composantes fortement connexes dans un graphe, on peut citer l'algorithme de Tarjan [63]. et celui de Kosaraju-Sharir [61] connus pour être des plus efficaces. Néanmoins, le fait qu'ils soient basés sur un parcours en profondeur d'abord (Depth First Search) les rend très mal adaptés à la distribution.

Définition 16 *Soit x un sommet, x^* désigne l'ensemble des sommets atteignables à partir de x, appelé aussi l'ensemble des sommets* descendants *de x. x^- désigne l'ensemble de sommets à partir desquels x peut être atteint, appelé aussi l'ensemble des sommets* ascendants *de x.*

Lemme 1 *Soit $G = (V, E)$ un graphe. La composante fortement connexe à laquelle un sommet x appartient est définie par $C^x = x^* \cap x^-$.*

A partir de ce lemme, on peut envisager l'utilisation de la stratégie *diviser pour conquérir* décrite dans l'algorithme suivant :

Procedure 4 CFC(V,E)

begin
  if $V = \emptyset$ **then return**
  `choisir` $x \in V$
  `calculer` x^* et x^-
  `afficher` CFC $C^x = x^* \cap x^-$
  $V_1 := V - x^*$;
  $V_2 := V - x^-$;
  $V_3 := V - (x^* \cup x^-)$
  `CFC` (V_1, E) ;
  `CFC` (V_2, E) ;
  `CFC` (V_3, E)
end

Le principe de cet algorithme consiste à :

- choisir un sommet x n'appartenant à aucune composante fortement connexe parmi celles qui sont déjà construites,
- le marquer à " $*$ ".
- marquer tous les sommets descendants de x à " $*$ " puis tous les ascendants de x à " $-$ ".
- Les sommets ayant été marqués à la fois par " $*$ " et " $-$ " seront considérés comme appartenant à la même composante fortement connexe.
- On recommence ce processus jusqu'à ce que tous les sommets soient inclus dans une composante fortement connexe.

Pour construire de manière distribuée les différentes composantes fortement connexes, l'algorithme doit être adapté, étant donné que le graphe des marquages est réparti sur les différentes stations. Ainsi, si l'on a déterminé les $k-1$ composantes fortement connexes $C^1, C^2, \ldots, C^{k-1}$, pour construire la composante C^k, le processus $i = 1$ détermine le sommet x par lequel commence la procédure de marquage. Si tous les sommets du processus 1 appartiennent déjà à une composante fortement connexe, c'est le processus 2 qui cherchera parmi ses sommets celui qui n'est pas dans une composante fortement connexe, si tous les sommet du processus 2 appartiennent aussi à des composantes fortement connexes ça sera le processus 3 et ainsi de suite. Ce rôle est transmis d'un processus à l'autre par envoi d'un message, et le processus garde ce rôle jusqu'à ce que tous les sommets qui lui appartiennent soient dans des composantes fortement connexes.

La procédure de marquage opère en deux phases :

1. Lors de la première phase on commence par le marquage de x et de tous les sommets locaux descendants de x à $*$. Des messages contenant $(x', *)$ sont envoyés aux processus propriétaires de chaque successeur x' distant. Lorsqu'un processus reçoit un message pour marquer un sommet x' à $*$, il le marque et marque aussi ses descendants locaux. Il envoie éventuellement des messages lorsqu'un prédécesseur appartient à un autre processus.

2. Dans la seconde phase, le sommet x est marqué à $-$ puis tous les ascendants locaux ayant déjà été marqués à $*$ sont marqués à $-$. De même, des messages $(x', -)$ sont envoyés au processus contenant les prédécesseurs distants des sommets marqués à $-$.

Le fait de ne considérer que les sommets ayant déjà été marqués à $*$ dans la deuxième phase permet de réduire le nombre de messages échangés.

Les sommets marqués à la fois par $*$ et $-$ constituent alors la composante fortement connexe C^k. En effet, si en partant d'un sommet x on marque un autre sommet y à $*$ (par un parcours avant) cela signifie qu'il existe un chemin menant de x à y, et si on marque y à $-$ (par un parcours arrière) cela signifie qu'il existe aussi un chemin de y vers x cela implique donc que x et y appartiennent à une même composante fortement connexe.

3.7 Expérimentations

Les algorithmes de vérification présentés ont été implémentés et testés sur un cluster. Le prototype ainsi réalisé permet d'analyser des réseaux de Petri simples, et vérifier en particulier les propriétés de vivacité et de l'état d'accueil. Les résultats des tests effectués sur les deux problèmes classiques, à savoir le problème des philosophes et le problème des gestionnaires de bases de données distribuées sont donnés respectivement par les tables 3.1 et 3.2. La dernière colonne indique le temps nécessaire lorsqu'on n'utilise qu'un seul processus (cas séquentiel). Les temps donnés dans ces tableaux correspondent à la moyenne des cumuls du temps de génération de l'espace d'états, de vérifcation de la vivacité d'une transition donnée et si un état donné est un état d'accueil.

Nbre Phil	Nbre d'états	Nb Trans. (N1)	Nb Trans Trav (N2)	N1/N2	Nb Mess. (N3)	N3/N2	tps CPU (sec)	tps CPU 1 proc
5	11	30	16	1.88	152	9.5	<0.01	<0.01
10	123	680	200	3.40	1,144	5.72	<0.01	0.01
15	1,364	11,310	2,260	5.00	11,535	5.10	0.06	0.12
20	15,127	167,240	25,084	6.67	125,815	5.02	0.87	3.85
25	167,761	2,318,400	278,206	8.33	1,391,691	5.00	44.14	521.50

TABLE 3.1 – Philosophers Problem

Pour le problème des philosophes, on peut remarquer que le rapport entre le nombre d'arcs traversants et le nombre d'arcs total, donné par N_1/N_2, croit pour atteindre une valeur proche de 8, ce qui signifie que l'on a un arc traversant pour huit arcs locaux. Ce ratio est raisonnable et permet de limiter le nombre de messages échangés. Ce nombre étant de 5 messages pour chaque arc traversant, comme cela est indiqué par le ratio N_3/N_2.

Nb Managers	Nbre d'états	Nbre de Trans.	Nb Trans. Trav.	Nb Mess.	tps CPU (Sec)	tps CPU 1 proc
6	1,460	4,878	14	208	<0.01	<0.01
7	5,105	20,433	15	224	0.17	0.01
8	17,498	81,688	16	245	1.38	2.31
9	59,051	314,955	19	282	2.43	23.18
10	196,832	1,181,010	22	293	117.73	283.12

TABLE 3.2 – Problème des gestionnaire des bases de données distribuées

En ce qui concerne le problème des gestionnaires de bases de donnèes distribuèes, malgré une taille relativement importante de l'espace d'états le nombre de messages n'augmente que légèrement, et cela toujours grâce à la fonction de répartition choisie, garantissant un nombre très réduit d'arcs traversants.

3.8 Conclusion

Dans ce chapitre nous avons présenté des algorithmes de vérification distribués pour des propriétés dites générales, telles que l'atteignabilité, le blocage, la vivacité et l'état d'accueil. La première est basée sur le parcours de l'espace d'états réparti à la recherche d'un état donné, tandis que la seconde elle consiste à rechercher des états n'ayant pas de successeur. Pour ces deux proprietés on peut espérer une accélération linéaire. Nous avons aussi préseté la démarche nécéssaire pour exhiber le chemin le plus court menant à un état considéré comme non souhaitable ou un état d'erreur. Les troisième et quatrième propriétés nécessitent la détermination des composantes fortement connexes du graphe des marquages. L'algorithme basé sur la politique diviser pour régner est le plus adéquat au cas distribué.

Chapitre 4

Model-checking LTL distribué

4.1 Introduction

Les premiers algorithmes de model-checking [55, 21, 47] sont apparus dans les années 80, et ont servi à l'élaboration de multiples outils de vérification. Avec l'amélioration des techniques utilisées, le model-checking s'est vite développé et est maintenant utilisé à large échelle, en particulier dans l'industrie pour la vérification de systèmes critiques.

Le model-checking LTL consiste à rechercher des cycles acceptants dans le graphe d'un automate de Büchi. C'est le principe utilisé pour la vérification des propriétés exprimées en logique temporelle linéaire. Il a été implémenté dans plusieurs outils de vérification (SPIN [37], Murφ [25], SVM [48], UPAAL [7], ...).

Adapter cette méthode au cas distribué a aussi fait l'objet de plusieurs travaux de recherche [45, 6]. Dans ce chapitre, nous présentons comment les algorithmes de vérification ont été adaptés au cas où l'espace d'état est distribué, les difficultés rencontrées et les solutions proposées pour pallier à ces difficultés, mais avant cela nous rappelons d'abord les principes de base du model-checking LTL.

Définition 17 *(Automate de Büchi)*
Un automate de Büchi est un quintuplet $\mathcal{A} = (S, S_0, \Sigma, \Delta, F)$ *où :*

- S *est un ensemble fini d'états*
- $S_0 \subset S$ *est l'ensemble des états initiaux*
- Σ *est l'ensemble de l'alphabet*

- $T \subset S \times \Sigma \times S$ *est un ensemble de transitions*
- $F \subset S$ *est l'ensemble des états d'acceptation*

L'ensemble des mots reconnus par un automate de Büchi est l'ensemble des mots définissant un chemin partant d'un état initial et visitant F (ensemble des états d'acceptation) une infinité de fois. Cet ensemble est noté $\mathcal{L}_\omega(\mathcal{A})$.

4.2 Model-checking LTL

Le model-checking LTL vise à vérifier des propriétés exprimées par des formules de la logique temporelle linéaire LTL sur le modèle opérationnel du système. L'idée est alors de construire un automate correspondant à la négation de formule LTL qui décrit la propriété. Les automates à construire sont généralement des automates de Büchi (automate à états) [67].

Etant donné un système S et une propriété φ, le model-checking renvoie après vérification le résultat *vrai* si le système S vérifie la propriété φ et *faux* sinon.

Les algorithmes les plus courants sont basés essentiellement sur les quatre points suivants :

- **Générer l'espace d'états du système à vérifier** : Ce dernier peut être décrit par un système de transitions étiquetées, une structure de Kripke, ou un automate qu'il faut ensuite transformer en un automate de Büchi. Soit A_M cet automate.
- **Construire l'automate correspondant à la propriété à vérifier** : l'automate utilisé correspond en fait à la négation de la formule. Soit $A_{\neg\varphi}$ l'automate construit. La transformation de la formule LTL en automate se fait à l'aide de la méthode dite méthode de tableaux [46].
- Calculer **le produit synchronisé de** A_M **et** $A_{\neg\varphi}$ ($A_M \otimes A_{\neg\varphi}$) : il s'agit là de déterminer l'intersection des langages acceptés par A_M et par $A_{\neg\varphi}$.
- **Tester le vide du langage reconnu par** $A_M \otimes A_{\neg\varphi}$: Cela consiste à appliquer un algorithme d'exploration pour détecter et calculer le ou les cycles acceptants. Ainsi, si aucun cycle acceptant n'est trouvé, la propriété est satisfaite, sinon le chemin menant à un cycle acceptant correspond à un contre-exemple.

La figure 4.1 donne les étapes de la vérification (Model-Checking) d'une propriété exprimée en LTL.

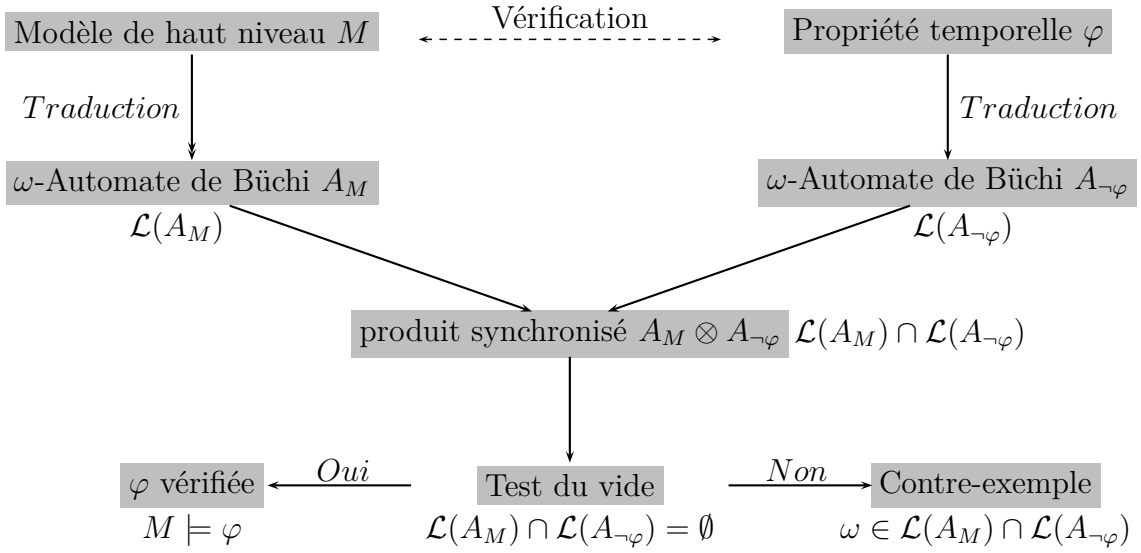

FIGURE 4.1 – Processus de vérification de formules LTL.

4.2.1 Test du vide d'un automate de Büchi

Ce test consiste à vérifier si le langage d'un automate de Büchi est vide. Pour cela, il existe plusieurs méthodes dont le principe commun est d'essayer de trouver un mot infini sous la forme uv^ω accepté par l'automate. Le mot u correspond au chemin d'accès d'un état initial à l'état répété et le mot v à la séquence répétée.

La méthode la plus courante consiste en une double exploration en profondeur (Nested Depth First Search) dans l'automate, décrite pour la première fois dans [24, 29], qui permet de chercher l'ensemble des états acceptants qui appartiennent à des cycles. Le langage de l'automate est non vide si et seulement si un de ces états est rencontré.

La deuxième méthode est basée sur la détermination des composantes fortement connexes accessibles de l'automate, utilisant l'algorithme de Tarjan [63] par exemple. Le langage de l'automate est non vide si et seulement si l'une des composantes fortement connexes accessibles contient un état acceptant.

4.2.2 De la logique temporelle linéaire aux automates

Un des points centraux de l'algorithme général du model-checking LTL, que nous venons de présenter, est la transformation d'une formule de logique temporelle en un automate qui reconnaît son langage. Il en résulte un intérêt marqué de la recherche dans ce domaine. Le point de départ est assez clairement le travail de Büchi. Une propriété de la logique temporelle linéaire peut être traduite en automates de Büchi. Dans [69], il a été prouvé que tous les langages ω-réguliers sont reconnaissables par un automate fini appelé automate de Büchi, et un algorithme de construction de l'automate correspondant à une formule LTL y a été défini.

La construction de l'automate de Büchi se fait avant de procéder au model-checking. L'avantage est qu'on peut réduire le graphe de l'automate en supprimant les états dupliqués avant de commencer la vérification.

Nous ne décrivons pas ici l'algorithme de construction de l'automate de Büchi d'une formule LTL. Nous nous contentons de donner quelques exemples.

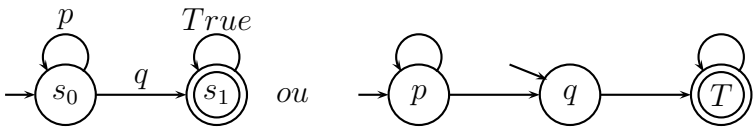

FIGURE 4.2 – Automate de Bûchi de la formule pUq

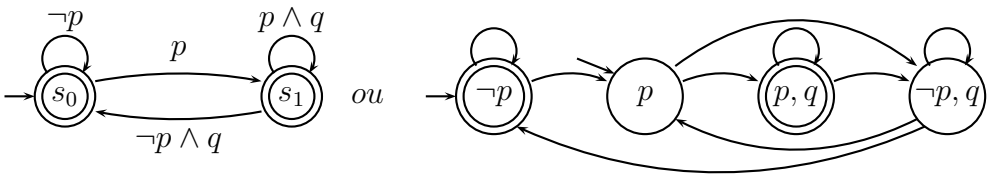

FIGURE 4.3 – Automate de Bûchi de la formule $G(p \Rightarrow Xq)$

4.2.3 Algorithmes de détection des cycles acceptants

Le problème de model-checking se réduit à la question de savoir si le produit synchonisé $A_M \otimes A_{\neg\varphi}$ admet ou non une exécution acceptée. Une hypothèse importante est que A_M

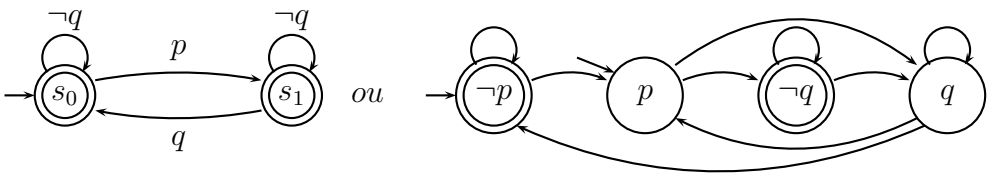

FIGURE 4.4 – Automate de Bûchi de la formule $G(p \Rightarrow Fq)$

doit être fini en termes de nombre d'états et de transitions.

Si l'on considère ce produit synchronisé comme un graphe dont les nœuds sont les états et les arcs sont ses transitions, cela revient à vérifier s'il existe un cycle dans ce graphe qui soit accessible depuis l'état initial et qui contient un état acceptant.

Deux algorithmes peuvent être utilisés :

1. L'algorithme de Tarjan [63] pour calculer l'ensemble des composantes fortement connexes accessibles depuis l'état initial. On cherche ensuite parmi ces composantes fortement connexes si l'une d'elle contient un état acceptant. Cette méthode reste peu utilisée car elle est incompatible avec certaines techniques d'optimisation ;

2. L'algorithme Nested Depth First Search (implémenté dans SPIN). Il s'agit d'un parcours du graphe en profondeur d'abord depuis l'état initial pour trouver dans un premier temps tous les états acceptants accessibles. Si aucun état acceptant n'est trouvé dans cette étape, c'est que la propriété est satisfaite. Sinon, un deuxième parcours en profondeur du graphe depuis chaque état acceptant s accessible est fait de façon à détecter s'ils sont accessibles depuis eux-mêmes c'est-à-dire qu'ils appartiennent à un circuit.

Ces algorithmes se terminent puisque par hypothèse le produit synchronisé est fini. Le pseudo-code de cette procédure est le suivant :

La procédure $DFSblue$ visite en profondeur d'abord tous les états atteignables et lance la procédure $DFSred$ lorsqu'un état d'acceptation est rencontrée. La procédure $DFSred$ vérifie si le sommet appartient à un cycle.

Il est a noté que la procédure $DFSred$ ne visite les sommets qu'une seule fois, le temps d'exécution est donc linéaire.

Procedure 5 $DFSblue(s)$

```
begin
    s.blue := true
    forall the t ∈ succ(s) do
        if ¬t.blue then
            DFSblue(t)
    if s ∈ Accepting then
        seed = s
        DFSred(s)
end
```

Procedure 6 $DFSred(s)$

```
begin
    s.red := true
    forall the t ∈ succ(s) do
        if t = seed then
            /* t est un état acceptant appartenant à un cycle */
            ExitCycle
        if ¬t.red then
            DFSred(t)
end
```

4.2.4 Limites du model-checking

Le model-checking est la méthode la plus complète et la plus utilisée dans le domaine de la vérification formelle des systèmes critiques et complexes. Elle est complètement automatique et offre des possibilités pour vérifier plusieurs types de propriétés (propriété générales, logiques, ...). Mais l'explosion combinatoire de l'espace d'états reste le problème principal pour appliquer la méthode sur des systèmes de grandes taille. En effet, par construction, la taille du produit synchronisé est exponentielle, ce qui limite en pratique l'utilisation du model-checking comme technique de vérification : le produit à construire est souvent trop

grand pour pouvoir être stocké en mémoire. Pour pallier à cet inconvénient, des techniques de construction et d'exploration partielle du produit synchronisé ont été mises au point.

4.2.5 La vérification à la volée

Il n'est pas conseillé de construire la totalité du résultat du produit synchronisé $A_M \otimes A_{\neg\varphi}$ et de le garder en mémoire pour ensuite effectuer le test du vide. La vérification à la volée (on-the-fly) consiste alors à vérifier l'existence d'états acceptants, appartenant à un cycle, en même temps que la génération de l'automate produit $A_M \otimes A_{\neg\varphi}$. On arrête la génération dès qu'on trouve un tel état.

Remarque 2 *Lors de la vérification à la volée :*

- *Seul le chemin courant est gardé en mémoire.*
- *On ne gagne rien si* $\mathcal{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$
- *Si* $\mathcal{L}(A_M \otimes A_{\neg\varphi}) \neq \emptyset$ *l'algorithme est utilisable sur des systèmes de transition infinis.*

D'autres techniques telles que d'ordre partiel exploitent la nature concurrente des systèmes pour n'explorer que certaines exécutions.

Parallèlement, des méthodes de model-checking symbolique ont été développées pour représenter des ensembles d'états plutôt que chaque état explicitement ; les transitions sont alors réexprimées sur ces états symboliques. Les diagrammes de décision binaires sont un exemple d'une telle technique.

Nous décrirons dans la suite l'approche basée sur la distribution comme solution à ce problème de l'éxplosion combinatoire.

4.3 Model-checking LTL distribué

L'algorithme séquentiel du model-checking LTL, qui a été présenté, est basé sur la recherche en profondeur d'abord (DFS). Le point important dans ce type de recherche est qu'elle préserve l'ordre de visite des états (post-ordre). Ce point est utilisé dans l'algorithme séquentiel pour la détection des cycles acceptants. Le problème de l'adaptation de l'algorithme séquentiel au cas distribué est que l'ordre DFS n'est plus préservé. Cela peut être illustré par le scénario suivant.

Si, pour l'exemple de la Figure 4.5, deux procédures Nested DFS s'exécutent simultanément au niveau des états A et B, le cycle passant par B peut ne pas être détecté.

Le résultat dépend de l'ordre de visite des états par les deux procédures Nested DFS. En effet, si la procédure Nested DFS commençant à partir de l'état A visite l'état C avant la procédure commençant à partir de B, cette dernière ne va pas continuer l'exploration à travers l'état C. Par conséquent, le cycle B, C, D, B ne sera jamais détecté.

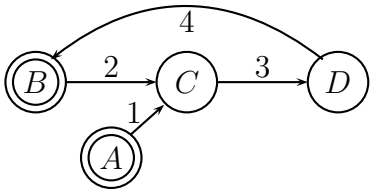

FIGURE 4.5 – Exécution incorrecte de la procédure DFS dans le cas parallèle.

Les procédures Nested DFS peuvent engendrer des résultats incorrects dans le cas distribué, si elles sont adaptées de manière naïve.

D'une façon générale, lorsque les parties de l'espace d'états explorées par deux procédures Nested DFS ont une intersection non vide, il y a une possibilité de ne pas détecter des cycles.

Pour pallier à ce problème, une solution évidente consiste à garder pour chaque nœud l'ensemble des chemins ayant permis à la procédure DFS de l'atteindre. Mais cela signifie qu'il faut stocker une quantité non négligeable d'informations, particulièrement lorsqu'il y a plusieurs processus exécutant la procédure DFS en parallèle.

Une autre approche basée sur le calcul des composantes fortement connexes, peut être utilisée en distribué pour le test du vide dans un automate. En effet, comme nous l'avons précisé précédemment, l'algorithme de Kosaraju-Sharir [61] est bien adapté pour un calcul distribué des composantes fortement connexes d'un graphe. Ainsi, le test consiste à seulement vérifier si un état d'acceptation se trouve dans une composante fortement connexe non triviale atteignable. Néanmoins, cette approche reste peu adaptée à des optimisations de calcul.

Nous allons décrire dans ce qui suit quelques approches proposées dans la littérature permettant aussi de pallier à ce problème.

4.3.1 Nested DFS distribuée avec structures de données supplémentaires

Dans SPIN [45], la procédure Nested DFS se fait en deux étapes. Lors de la première, le processus de recherche en profondeur d'abord (DFS) des états acceptants est effectué en parallèle, et les états acceptant rencontrés sont gardés en mémoire avec des informations relatives à leurs dépendances dans une structure appelée *structure des dépendances*. La recherche de cycles à partir de ses états se fait séparément selon un ordre approprié défini par les différentes dépendances. Durant cette deuxième étape, une seule procecédure Nested DFS est exécutée à la fois.

La procédure Nested DFS n'est autorisée à examiner si un état d'acceptation appartient à un cycle que si les états qui le suivent dans la structure de dépendance ont déjà été examinés.

La structure de dépendance permet de connaître les états impliqués dans le transfert du calcul d'un nœud à un autre ainsi que les états d'acceptation rencontrés durant le processus d'exploration.

Chaque processus i construit pendant l'exploration DFS, de manière dynamique, sa structure de dépendances, qui consiste en une forêt composée des sommets comprenant les états d'acceptation appartenant au nœud i et les états dits de transfert. L'état initial appartient à la structure de dépendance du nœud comprenant cet état.

Un état s est un état de transfert du processus i si s appartient au processus i et possède un prédécesseur distant qui est aussi un état de transfert du nœud auquel il appartient. L'état s est alors indexé par les numéros des processus par lesquels le calcul a été transferé au nœud i. Un état s distant peut aussi être état de transfert du processus i s'il possède un prédecesseur dans i.

La structure de dépendance est construite de manière dynamique durant la procédure DFS. Si la procédure DFS rencontre un état d'acceptation qui n'a pas été visité auparavant ou un état de transfert, un nouvel état est alors ajouté dans la structure de dépendance comme successeur du dernier état visité présent dans la structure. De plus, dans le cas d'un état de transfert distant un message est envoyé au processus propriétaire (s'il n'a pas déjà été envoyé auparavant) pour poursuivre l'exploration.

Si un processus i reçoit un message d'un processus j pour poursuivre l'exploration à

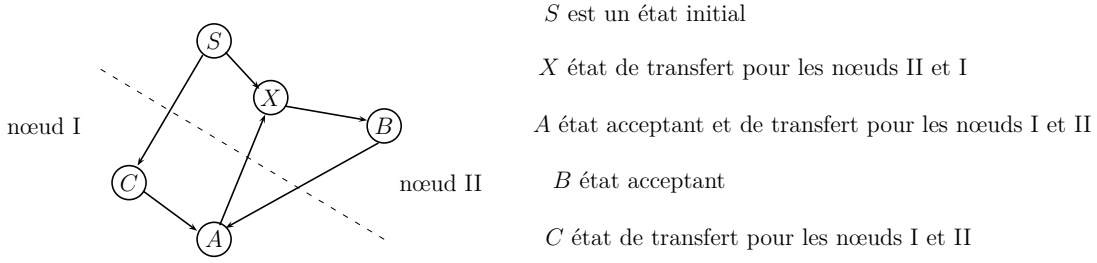

FIGURE 4.6 – Etats de transfert et états d'acceptation.

partir d'un état s, il vérifie alors :

- Si l'état s a déjà été exploré et existe aussi dans la structure de dépendance. j est alors ajouté comme index à l'état s.
- Si s n'a pas été exploré, s est alors ajouté comme racine dans la structure avec l'index j.
- Si s se trouve parmi les états visités mais pas dans la structure de dépendances, un accusé est renvoyé au processus j.

Si un état d'acceptation se trouve être une feuille dans la structure, cet état est envoyé au processus coordinateur, qui va l'ajouter dans la file d'attente pour la procédure Nested DFS, puis le supprimer de la structure de dépendance.

Lorsque tous les processus ont fini la phase d'exploration, chacun envoie au processus coordinateur des élements de la forme : $X_I \xrightarrow{u} Y$ où X_I est un état appartenant à la table de dépendances avec un index I non vide, et Y son successeur dans la structure. L'arc est étiqueté par 1 s'il y a entre X et Y (X inclus) un état d'acceptation et 0 sinon. La Figure 4.7 donne un exemple de construction de la structure de dépendances.

Le coordinateur détermine s'il y a une composante fortement connexe terminale ayant des arcs étiquetés avec des 1. Si tel est le cas, la formule n'est pas vérifiée, sinon l'exploration est poursuivie normalement.

4.3.2 Détection des cycles négatifs

En théorie des graphe, un cycle est dit négatif si le poids total de ses arcs est négatif. Le problème de détection des cycles acceptés peut être réduit au problème de détection de cycles négatifs. La relation avec les automates de Büchi est la suivante : un automate de Büchi correspond à un graphe, si nous assignons des longueurs aux transitions de manière

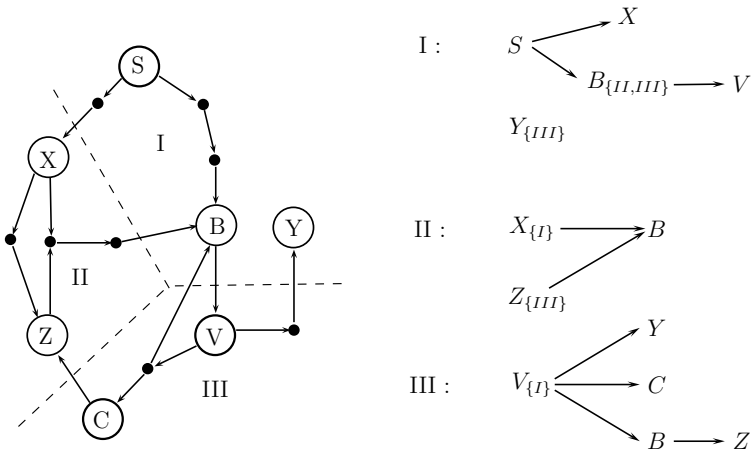

FIGURE 4.7 – Exemple d'une structure de dépendances.

que les transitions qui partent d'un état d'acceptation sont à -1, et toutes les autres sont à 0, alors les cycles négatifs coïncident avec les cycles d'acceptation. Ceci permet de réduire le problème de la détection du vide de l'automate de Büchi au problème de détection de cycles négatifs [14]. Ce dernier problème est complètement lié au problème du plus court chemin à partir d'un seul état source (SSSP : Single Source Shortest Path problem). Il s'agit de trouver les chemins menant d'un état source spécifique vers tous les autres états.

4.3.3 Détection des cycles basée sur la construction du graphe réduit

L'approche que nous proposons pour effectuer de manière distribuée le test du vide est basée sur la construction des graphes réduits locaux par chaque processus. Ainsi, chaque processus détermine toutes les composantes fortement connexes locales (en utilisant par exemple l'algorithme de Tarjan). On distinguera alors trois types de composantes fortement connexes :

- Celles ne contenant aucun état d'acceptation.
- Les composantes fortement connexes triviales, réduites à un état acceptant sans boucles.
- Les composantes fortement connexes non triviales contenant des états d'acceptation.

Dans ce dernier cas, on peut dores et déjà dire que le graphe global comporte un circuit acceptant et donc la formule n'est pas vérifiée.

Dans le cas où les composantes fortement connexes non triviales ne contiennent pas d'états d'acceptation, on procède à la construction d'un graphe contracté contenant des nœuds représentant les composantes fortement connexes des graphes locaux. Le graphe de dépendances contiendra uniquement les nœuds représentant la composante fortement connexe à laquelle appartient l'état initial et les composantes fortement connexes comportant des états de transfert. Rappelons qu'un état est dit état de transfert s'il est extrémité d'un arc traversant, c'est-à-dire un arc ayant une extrémité appartenant à un processus et l'autre extrémite à un autre processus. Le graphe contracté est construit, au niveau du coordinateur, à partir des informations envoyées par les autres processus, de la manière suivante :

- Le processus p_0 propriétaire de l'état initial commence la construction en envoyant les triplets (i^{p_0}, j^{p_0}, u) au processus coordinateur si les composantes fortement connexes C_i et C_j contiennent soit l'état initial soit un état extrémité d'un arc traversant et il existe un chemin reliant un sommet de C_i à un sommet de C_j ; u est égal à 1 si sur l'un des chemin il y a un état acceptant ou 0 sinon.
- Chaque autre processus p fait de même en envoyant les triplets (i^p, j^p, u) au processus coordinateur si les composantes fortement connexes C_i et C_j contiennent un état extrémité d'un arc traversant et il existe un chemin reliant un sommet de C_i à un sommet de C_j ; u est égal à 1 si sur l'un des chemin il y a un état acceptant ou 0 sinon.
- Pour chaque état r extrémité initiale d'un arc de transfert appartenant à la composante fortement connexe C_r le processus propriétaire p envoie un message (r, C_r, u) au processus ayant l'état successeur ; u est égal à 1 si la C_s contiant un état d'acceptation, à 0 sinon.
- Chaque processus p' recevant le message précédent du processus p, envoie au coordinateur le message $(C_r^p, C_s^{p'}, u)$ s'il n'a pas été déjà envoyé, C_s étant la composante fortement connexe contenant l'état successeur.

4.4 Conclusion

Dans ce chapitre, nous avons présenté les difficultés et les solutions apportées pour l'adaptation des algorithmes du model-checking séquentiel de la logique temporelle linéaire LTL à la distribution. Cette adaptation n'est pas triviale. Ces algorithmes se basent sur la procédure NDFS, qui n'est pas facile à adapter au cas distribué. Dans les solutions pro-

posées, il fallait introduire des informations supplémentaires afin de parvenir à la détection de cycles dans le cas d'un graphe réparti. Parmi ces solutions on trouve :

- l'utilisation de structures de dépendance.
- la détection de cycles négatifs.

Nous avons proposé une procédure pour vérifier l'existence d'un cycle accéptant basé sur une construction en parallèle des composantes fortement connexes des graphes locaux de chaque processus. Si parmi celles-ci, il y a des composantes fortement connexes non triviales qui contiennent des états d'acceptation, on peut conclure alors que la propriété vérifiée n'est pas valide, sinon un graphe contracté est construit au niveau du processus coordonnateur, qui vérifie à son tour s'il y a ou non des cycles acceptants dans ce graphe.

Chapitre 5

Model checking CTL distribué

5.1 Introduction

La logique CTL a été introduite par Clarke et Emerson [21]. Elle permet d'exprimer des propriétés aussi bien sur des chemins que sur les arbres d'exécution. C'est pourquoi elle est classée comme étant une logique temporelle arborescente, basée sur des modèles où à tout moment il peut y avoir plusieurs futurs possibles. La syntaxe des formules CTL est donnée dans la définition 1, où les expressions les plus élémentaires sont les propositions atomiques. L'ensemble des propositions atomiques est désigné par AP, et seses éléments sont généralement notés $p, q, r, \ldots$

Dans ce chapitre, nous nous intéressons à la vérification distribuée de propriétés exprimées en CTL sur un espace d'états réparti, et ce toujours dans le but d'analyser des systèmes de grande taille.

Définition 18 *(Syntaxe de CTL) Les formules de CTL sont définies comme suit :*

- $p \in AP$ *est une formule de CTL,*
- *Si* φ *et* ψ *sont des formules de CTL, alors* $\neg\varphi$, $\varphi \wedge \psi$, *et* $\varphi \vee \psi$ *sont des formules temporelles de CTL,*
- *Si* φ *et* ψ *sont des formules de CTL, alors* $EX\varphi$, $E(\varphi U \psi)$ *et* $A(\varphi U \psi)$ *et* $p_1 U p_2$, *sont des formules temporelles de CTL.*

Les symboles E et A sont des quantificateurs existentiel et universel. Ils expriment respectivement que la propriété est valide sur au moins un chemin ou pour tout chemin.

Les opérateurs X et U sont des opérateurs temporels, appelés respectivement *neXt* et *Until*. Ils doivent se trouver dans le champ d'un opérateur existentiel ou universel.

Les modèles de la logique temporelle sont définis sur des automates appelés structures de Kripke [21], définis comme suit.

Définition 19 *(Structure de Kripke) Une* structure de Kripke *est un quadruplet* $K = \langle S, s_0, T, \lambda \rangle$ *où :*

- S *est un ensemble non vide d'états,*
- s_0 *est l'état initial,*
- $T \subseteq S \times S$ *est l'ensemble des transitions, associant à chaque état* $s \in S$*, ses successeurs possibles.*
- $\lambda : S \longrightarrow 2^{AP}$ *associe à tout état* $s \in S$ *l'ensemble des propositions vraies dans* s*.*

Définition 20 *(Chemins) Un* chemin *est une séquence infinie d'états* $\sigma = s_1 s_2 s_3 \ldots \in S^\omega$ *tel que :* $\forall i \geq 1 \ (s_i, s_{i+1}) \in T$*.*

On note par $\sigma(i)$ l'élément i du chemin σ, et on désigne par $P_M(s)$ l'ensemble des chemins dont l'origine est s, i.e., $P_M(s) = \{\sigma \mid \sigma(0) = s\}$.

La séquence finie $s_1 s_2 \ldots s_n$ est dite préfixe de σ si $\sigma(i) = s_i, \forall 0 \leq i \leq n$.

Définition 21 *(Sémantique de CTL) Soient* $p \in AP$ *une proposition atomique,* $M = \langle S, s_0, T, \lambda \rangle$ *une structure de Kripke,* $s \in S$*, et* φ, ψ *deux formules CTL. La relation de satisfaction* $\models$ *est définie de manière inductive comme suit :*

$s \models p$	*ssi* $p \in \lambda(s)$
$s \models \neg\varphi$	*ssi* $\neg(s \models \varphi)$
$s \models \varphi \vee \psi$	*ssi* $(s \models \varphi) \vee (s \models \psi)$
$s \models EX\varphi$	*ssi* $\exists \sigma \in P_M(s).\sigma[1] \models \varphi$
$s \models E[\varphi \cup \psi]$	*ssi* $\exists \sigma \in P_M(s), \exists j \geq 0 : \sigma[j] \models \psi \wedge (\forall 0 \leq k < j : \sigma[k] \models \varphi)$
$s \models A[\varphi \cup \psi]$	*ssi* $\forall \sigma \in P_M(s) : \exists j \geq 0, \sigma[j] \models \psi \wedge (\forall 0 \leq k < j : \sigma[k] \models \varphi)$

L'interprétation des propositions atomiques, de la négation et de la conjonction sont classiques.

Quant à la formule $EX\varphi$, elle est valide si et seulement si il existe un chemin σ où s est l'état initial et son successeur dans σ vérifie φ.

La formule $A[\varphi \cup \psi]$ est valide dans l'état s si et seulement si tout chemin de $P_M(s)$ possède un préfixe (qui peut ne contenir qu'un seul état) dont le dernier état vérifie ψ et tous les autres états vérifient φ.

$E[\varphi \cup \psi]$ est valide dans s si et seulement si il existe un chemin appartenant à $P_M(s)$ possédant un préfixe dont le dernier état vérifie ψ et tous les autres états vérifient φ.

On dit que $M \models \phi$ si et seulement si $s_0 \models \phi$.

Des abréviations sont aussi définies et souvent utilisées dans la littérature :

$EF\varphi \equiv E(true \cup \varphi)$	"φ potentiellement"
$EG\varphi \equiv \neg AF\neg\varphi$	"toujours potentiellement φ"
$AF\varphi \equiv A(true \cup \varphi)$	"φ inévitablement "
$AG\varphi \equiv \neg EF\neg\varphi$	"invariablement φ"
$AX\varphi \equiv \neg EX\neg\varphi$	"pour tout chemin Next φ"

5.2 Le Model-Checking CTL

L'algorithme du model-checking pour la logique temporelle CTL consiste à calculer de manière itérative, pour une formule φ donnée et un modèle M le sous-ensemble d'états $Sat_M(\varphi)$ vérifiant φ, c'est-à-dire $Sat_M(\varphi) = \{s \in S \mid s \models \varphi\}$.

function $Sat(\varphi$:Formula) : **set of** States ;
begin
if $\varphi = true \longrightarrow$ return S
$\| \ \varphi = false \longrightarrow$ return $\emptyset$
$\| \ \varphi \in AP \longrightarrow$ return $\{s | \varphi \in \lambda(s)\}$
$\| \ \varphi = \neg\varphi_1 \longrightarrow$ return $S \setminus Sat(\varphi_1)$
$\| \ \varphi = \varphi_1 \vee \varphi_2 \longrightarrow$ return $(Sat(\varphi_1) \cup Sat(\varphi_2))$
$\| \ \varphi = EX\varphi_1 \longrightarrow$ return $Sat_{EX}(\varphi_1) = \{s \in S | \exists (s, s') \in R \wedge s' \in Sat(\varphi_1)\}$
$\| \ \varphi = E[\varphi_1 \cup \varphi_2] \longrightarrow$ return $Sat_{EU}(\varphi_1, \varphi_2)$
$\| \ \phi = A[\varphi_1 \cup \varphi_2] \longrightarrow$ return $Sat_{AU}(\varphi_1, \varphi_2)$
fi
end

La vérification de $s \models \varphi$ consiste alors à vérifier si $s \in Sat_M(\varphi)$. La fonction Sat est alors construite en étiquetant inductivement par φ les états dans lesquels φ est vérifiée. Les algorithmes séquentiels complets peuvent être trouvés par exemple dans [21]. Dans la plupart des cas, ces algorithmes sont relativement simples, excepté pour les fonctions Sat_{EX}, Sat_{EU} et Sat_{AU}, qui seront détaillées dans ce qui suit pour le cas distribué.

5.3 Model-Checking distribué des formules CTL

Dans cette partie, nous présentons les algorithmes de vérification de formules CTL sur un espace d'états distribué. Chaque processus ayant une copie de la formule commence par décomposer cette formule pour obtenir toutes les sous formules possibles, et entamer ensuite la procédure d'étiquetage.

Les formules propositionnelles, les propositions atomiques, la négation et la conjonction peuvent être vérifiées localement par chaque processus. Cependant, lorsque la sous formule inclut des opérateurs temporels EX, EU ou AU, des communications entre les différents processus pour examiner les sommets successeurs ou prédécesseurs deviennent nécessaires (le traitement des messages sera détaillé dans le section 5.3.6. Ainsi, chaque processus i détermine l'ensemble $Sat_i(\varphi)$ des états satisfaisant la formule φ :

- $Sat_i(\varphi) \subseteq Sat(\varphi)$
- $\bigcup_{i=0..N-1} Sat_i(\varphi) = Sat(\varphi)$

5.3.1 Formules propositionnelles

Lorsque φ est une formule propositionnelle de CTL, tous les processus calculent indépendemment les uns des autres leurs ensembles $Sat_i(\varphi)$ qui satisfont la formule φ utilisant le pseudo-code donné par l'algorithme 7. Lorsque φ est elle-même composée de sous-formules, les ensembles d'états satisfaisant ces sous-formules sont supposés déj calculés.

Puisque il n'est nullement besoin de communications pour déterminer l'ensemble des états satisfaisant les formules propositionnelles de CTL, on peut espérer une accélération linéaire.

Procedure 7 `Sat`(i, φ, S_i)

```
begin
    Sat_i(φ) ← ∅
    switch φ do
        case p :
            foreach s ∈ S_i do
                if p ∈ λ(s) then Sat_i(φ) ← Sat_i(φ) ∪ {s}
        case ¬φ_1 :
            foreach s ∈ S_i do
                if s ∉ Sat_i(φ_1) then Sat_i(φ) ← Sat_i(φ) ∪ {s}
        case φ_1 ∧ φ_2 :
            foreach s ∈ S_i do
                if s ∈ Sat_i(φ_1) ∩ Sat_i(φ_2) then Sat_i(φ) ← Sat_i(φ) ∪ {s}
end
```

5.3.2 Cas de formules CTL $\phi = EX(\varphi)$

Pour vérifier des formules CTL de type $\phi = EX(\varphi)$, nous supposons que pour chaque processus i, l'ensemble des états vérifiant la formule φ est connu. Ainsi, pour déterminer l'ensemble des états vérifiant $\phi = EX(\varphi)$, chaque station exécute l'algorithme 8.

Procedure 8 `Sat`$(i, \phi = \mathbf{EX}\varphi, S_i)$

```
begin
    Sat_i(φ) ← ∅
    foreach s' in Sat_i(φ) do
        foreach predecessor s of s' do
            if h(s) = i then
                Sat_i(φ) ← Sat_i(φ) ∪ {s}
            else
                Send (h(s), < Sat_EX, s >)
end
```

Chaque processus affecte l'étiquette aux états locaux, prédécesseurs des états satisfaisant φ, et envoie un message aux processus auxquels appartiennent les prédécesseurs distants pour qu'ils soient aussi étiquetés. Le nombre de messages échangés sera donc au

plus égal au nombre d'arcs traversants. Aucune synchronisation n'est ncessaire, et il n'y a donc pas d'attente (inactivit) de la part des diffrents processus. On peut alors espérer aussi une accélération linéaire.

5.3.3 Cas de formules CTL $\phi = AX(\varphi)$

Pour la vérification de la formule CTL $\phi = AX(\varphi)$, il faut associer un compteur $HoldSuccNb(s)$ à chaque sommet s, qui permettra de savoir le nombre de successeurs de s qui satisfont φ. Ainsi, lorsque ce compteur atteint le degré sortant du sommet s, cela voudra dire que s satisfait ϕ, s est alors ajouté à $Sat_i(\phi)$ (voir l'algorithme 9).

La satisfaction de cette formule peut aussi être vérifiée d'une autre manière, qui consiste à vérifier la formule équivalente $\phi = \neg AX(\neg\varphi)$ en utilisant donc l'algorithme précédent. Néanmoins la vérification directe s'avère plus efficace.

Procedure 9 `Sat`$(i, \phi = \mathbf{AX}\varphi, S_i)$

```
begin
    Sat_i(φ) ← ∅
    foreach s' in Sat_i(φ) do
        foreach predecessor s of s' do
            if h(s) = i then
                HoldSuccNb(s) + +
                if HoldSuccNb(s) = OutDeg(s) then
                    Sat_i(φ) ← Sat_i(φ) ∪ {s}
            else
                Send (h(s), < Sat_AX, s >)
            /* Send a message to label non-local predecessors */
end
```

5.3.4 Cas de la formule CTL : $\phi = \mathbf{E}(\varphi_1 \cup \varphi_2)$

Pour évaluer la formule CTL $\phi = E(\varphi_1 \cup \varphi_2)$, nous supposons que $Sat_i(\varphi_1)$ et $Sat_i(\varphi_2)$ sont déjà connus pour chaque processus i. Ainsi, pour calculer l'ensemble $Sat_i(\phi)$, chaque processus exécute l'algorithme 10.

Comme dans le cas séquentiel, chaque processus i construit l'ensemble $Sat_i(\phi)$ des états satisfaisant la formule ϕ, qui au début est initialisé à $Sat_i(\varphi_2)$, puis chaque prédécesseur

d'un élement de $Sat_i(\phi)$ satisfaisant φ_1 est ajouté dans $Sat_i(\phi)$. Si le prédécesseur est distant, un message est alors envoyé au processus propriétaire j pour qu'il le rajoute dans son ensemble $Sat_j(\phi)$.

Procedure 10 `Sat`$(i, \phi = E(\varphi_1 \cup \varphi_2), S_i)$

```
begin
    Sat_i(φ) ← Sat_i(φ_2)
    Snew_i ← Sat_i(φ_2)
    foreach s' ∈ Snew_i do
        foreach predecessor s of s' do
            if h(s) = i then
                if s ∈ Sat_i(φ_1) \ Sat_i(φ) then
                    Snew_i ← Snew_i ∪ {s}
                    Sat_i(φ) ← Sat_i(φ) ∪ {s}
            else
                Send (h(s), < Sat_EU, s >)
end
```

5.3.5 Cas de la formule CTL $\phi = A(\varphi_1 \cup \varphi_2)$

Une idée similaire est utilisée pour déterminer l'ensemble de satisfaction de la formule CTL $\phi = A(\varphi_1 \cup \varphi_2)$. $Sat_i(\varphi_1)$ et $Sat_i(\varphi_2)$ étant connus, $Sat_i(\phi)$ est calculé par chacun des processus i. Des échanges de messages sont nécessaires lorsque des nøeuds voisins ne se situent pas sur la même station.

La détermination de l'ensemble de satisfaction $Sat_i(\phi)$ nécessite que chaque processus ait déjà déterminé $Sat_i(\varphi_1)$ et $Sat_i(\varphi_2)$. Cela implique que certains processus peuvent se mettre en attente inactive si $Sat_i(\varphi_1)$ et $Sat_i(\varphi_2)$ ne sont pas encore entièrement déterminés. Ainsi, une synchronisation (rendez-vous) est nécessaire avant de pouvoir déterminer l'ensemble de satisfaction de formules comportant l'opérateur temporel $E\cup$ ou $A\cup$ et uniquement dans ces cas.

5.3.6 Traitement des messages

La fonction `MessHandler` est étendue de manière à pouvoir traiter aussi les messages concernant l'évaluation des formules de la logique temporelle CTL comme décrit dans 12.

Procedure 11 $\texttt{Sat}(i, \phi = A(\varphi_1 \cup \varphi_2), S_i)$

```
begin
    Sat_i(φ) ← Sat_i(φ_2)
    Snew_i ← Sat_i(φ_2)
    foreach s' ∈ Snew_i do
        foreach predecessor s of s' do
            if h(s) = i then
                if s ∉ Sat_i(φ) ∧ s ∈ Sat_i(φ_1) then
                    HoldSuccNb(s) + +
                    if HoldSuccNb(s) = OutDeg(s) then
                        Snew_i ← Snew_i ∪ {s}
                        Sat_i(φ) ← Sat_i(φ) ∪ {s}
            else
                Send (h(s), < Sat_AU, s >)
end
```

Lorsqu'un processus i reçoit un message étiqueté par Sat_{EU}, le sommet s est ajouté à l'ensemble $Snew_i$ et la fonction $\texttt{Sat}(i, \phi = \mathbf{EX}\varphi, S_i)$ est ensuite appelée. Lorsqu'il reçoit Sat_{AU}, il incrémente $HoldSuccNb(s)$, car un nouveau successeur satisfait φ. Si $HoldSuccNb(s)$ atteint le degré sortant de s, alors s est ajouté à l'ensemble $Sat_i(\phi)$.

La terminaison se produit lorsque l'état initial s_0 est étiqueté par la formule à vérifier φ, ce qui correspond au cas où $M \models \phi$, ou bien lorsque tous les processus ont fini d'étiqueter tous leurs états. Dans ce cas le processus ayant en sa possession l'état s_0 conclut si ϕ est valide ou pas pour le modèle donné, selon que cet état est étiqueté ou pas par ϕ.

5.4 Recherche distribuée de contre-exemple

L'algorithme de model-checking se termine soit avec succès, si le modèle satisfait la formule ou la propriété souhaitée, soit par un échec si la formule n'est pas satisfaite. Dans ce cas, un contre-exemple permet de connaître les raisons de cet échec et peut aider à déterminer les erreurs dans des systèmes complexes. Une large classe de logique temporelle admet un sous graphe "Like-tree" comme contre-exemple [23], qui consiste en une structure qui prouve que la négation de la formule est vraie. Nous proposons ici de déterminer, sur un espace d'états distribué, le contre exemple lorsque la formule à vérifier n'est pas valide.

Procedure 12 `MessageHandler`(i)

```
begin
    ...  case Message.Type = Sat_EX
        Sat_i(φ) ← Sat_i(φ) ∪ {Message.State}
    case Message.Type = Sat_AX
        s ← Message.State
        HoldSuccNb(s) + +
        if HoldSuccNb(s) = OutDeg(s) then
            Sat_i(φ) ← Sat_i(φ) ∪ {s}
    case Message.Type = Sat_EU
        s ← Message.State
        if ((s ∉ Sat_i(φ) ∧ s ∈ Sat_i(φ_1)) then
            Snew_i ← Snew_i ∪ {s}
            Sat_i(φ) ← Sat_i(φ) ∪ {s}
        Sat(i, φ = EXφ, S_i)
    case Message.Type = Sat_AU
        s ← Message.State
        if (s ∉ Sat_i(φ) ∧ s ∈ Sat_i(φ_1)) then
            HoldSuccNb(s) + +   if HoldSuccNb(s) = OutDeg(s) then
                Snew_i ← Snew_i ∪ {s}
                Sat_i(φ) ← Sat_i(φ) ∪ {s}
        Sat(i, φ = AXφ, S_i)
    ...
end
```

Cela nous permettra d'utiliser les techniques d'énumération explicites des états du modèle, en exploitant les ressources de stations multiples connectées en réseau et communiquant par envoi de messages.

Dans cette partie, nous nous focalisons sur la détermination du contre-exemple pour des formules utilisant le quantificateur universel.

La recherche du contre-exemple peut être effectuée en profondeur d'abord, en commençant l'exploration à partir de l'état initial s_0. Il aisé de déterminer le contre-exemple correspondant à une formule de type $\phi = AX\varphi$. Lorsqu'une telle formule n'est pas vérifiée cela signifie simplement qu'un successeur s de s_0 ne vérifie pas φ, $s \not\models \varphi$. Ainsi le contre-exemple consiste en un sous graphe qui se limite seulement à l'état initial s_0 et ses successeurs qui ne vérifient pas φ.

Cependant, dans le cas d'une formule de la forme $A(\varphi_1 \cup \varphi_2)$, le contre exemple consiste en tous les chemins $\pi = s_0 s_1 \ldots$ tels que :

$$\exists i(i \geq 0 \wedge s_i \models (\neg\varphi_1 \wedge \neg\varphi_2) \wedge \forall j(0 \geq j < i \Rightarrow s_j \models (\varphi_1 \wedge \neg\varphi_2)))$$
$$\text{ou}$$
$$\exists i(i \geq 0 \wedge \exists j(j > 0 \wedge s_i = s_{i+j} \wedge \forall k(0 \leq k < i + j \Rightarrow s_k \models (\varphi_1 \wedge \neg\varphi_2)))$$

La première condition signifie que π comporte une suite d'états $s_0 s_1 \ldots s_{i-1}$ où φ est vérifiée, et dans l'tat s_i ni φ_1 ni φ_2 ne sont vérifiées.

Dans la seconde, le chemin π contient un cycle où φ_1 est toujours vérifiée mais jamais φ_2. Dans ce cas, il est possible que l'on exécute infiniment souvent le cycle sans atteindre un état où φ_2 est vérifiée. Pour extraire le sous-graphe correspondant au contre-exemple, on utilise l'algorithme 13.

Chaque processus marque ses états locaux, utilisant la pile V correspondant aux états déja visités et l'ensemble d'états P de sommets correspondant au chemin d'un éventuel contre-exemple en construction.

La détection de cycles peut être effectuée par une exploration en profondeur d'abord. Néanmoins, si cette exploration est effectuée en parallèle, la procédure de marquage peut mener à la détection de faux cycles. Une exploration séquentielle est donc adoptée en utilisant des primitives d'envoi de messages bloquantes.

Procedure 13 CounterExampleSearch()

```
begin
    while ¬Empty(V) do
        if TopStack(V) = CEBacktrack then
            Send (TopStack(V).Sender, < Backtrack >)
        else
            s ← TopStack(V)
            NoNewSucc ← true
            P ← P ∪ {s}
            foreach s' successor of s do
                if h(s') = i then
                    if s' ∈ M /* there is a cycle where φ1 holds */
                    then
                        Display(M) ∪ {s'}
                    else
                        if s' ∈ Sat_i(φ1) ∩ Sat_i(φ2); /* we get a path which is a
                           Counterexample */
                        then
                            Display(M) ∪ {s'}
                        if s' ∈ Sat_i(φ1) \ Sat_i(φ2) then
                            push (s', P)
                                M ← M ∪ {s'}
                                NoNewSucc = false; /* s has a succ satisfiying φ1 but
                                not ¬φ2 */
                else
                    Send (hash(s'), < s', CounterExampleCheck >)
            if NoNewSucc then
                if TopStack(P) ∈ M then
                    M ← M \ TopStack(P); Pop(P)
end
```

5.5 Implémentation et expérimentation

Les algorithmes précédents ont été implémentés sur un cluster de 12 stations [56]. Etant donné un réseau de Petri, son graphe des marquages accessibles est génér telle manire que les états soient répartis sur l'ensemble de stations. Nous avons considér là aussi les problèmes classiques des philosophes et des gestionnaires de bases de données réparties. Plusieurs formules CTL ont été analysées sur l'espace d'états réparti, et lorsqu'une formule n'est pas satisfaite, le sous-graphe correspondant au contre-exemple est alors déterminé.

Dans les cas des philosophes, nous avons vérifier des propriétés telles que :

- Le philosohe i fini toujours par se mettre à manger, qui est exprimée par $AGEF\, eating(i)$.
- De ux philosophes voisins mangent en même temps $EF(eating(i) \wedge eating(\text{i+1 mod n}))$.

La vérification des deux formules aboutit à un échec. Donc la recherche d'un contre-exemple est effectue. Le sous-graphe correspondant au contre-exemple de la première formule est une composante fortement connexe où le philosophe i n'est en train de manger dans aucun de ses états.

Les résultats des tests obtenus pour la vérification de la première formule sont donnés dans la table 5.1.

Nb Phil	Verif. time(sec)	1 proc verif time	Tot. Nb of CE	Paths CE	Mess. Nb 1^{st} CE	Tot. Nb Mess	Max cpu time
5	<0.01	<0.01	11	4	17	49	<0.01
10	<0.01	0.01	218	42	24	480	0.01
15	0.03	0.18	3,502	267	31	5,265	0.12
20	0.18	0.77	49,917	5,168	49	52,221	3.46
25	0.84	4.33	868,345	95,375	68	512,614	39.52
30	5.15	$\gg$	10.043,424	190,951	182	5,204,315	149.96

TABLE 5.1 – Vrification distribue et recherche de contre-exemple

Les premières colonnes de la table 5.1 donnent les résultats des tests concernant les temps CPU maximaux de chaque processus dans le cas distribué et le temps CPU dans le cas d'un seul processus (cas séquentiel). On remarque que l'accélération obtenue est relativement intéressante.

La deuxième partie des colonnes concerne la recherche distribuée des contre-exemples. On y trouve le nombre de contre-exemples, le nombre de contre-exemples qui sont des chemins (les autres sont des cycles), le nombre de messages qui ont été envoyés avant de

trouver le premièr contre-exemple et le nombre total de messages pour trouver tous les contre-exemples. Dans la dernire colonne on trouve le temps CPU maximum.

Pour le problème des gestionnaires de bases de données réparties, nous avons aussi considér des formules qui ne sont pas satisfaites, par exemple vérifier si un `manager` qui est dans l'état *Waiting* peut ne jamais recevoir d'acquittements *acknowledgment*. Les résultats des expérimentations sont illustrés dans la table 5.2.

Nb de Managers	Nb d'états	Nb de Trans	Nb d'arcs traversants	Nb Tot de Mess	Nb de chemins ds CE	Tps CPU de Génération	Tps CPU de vérification	Tps CPU 1 proc
6	1,459	4,878	14	24	112	0.02	< 0.01	0.01
7	5,104	20,433	16	28	184	0.04	< 0.01	0.77
8	17,497	81,688	18	32	282	0.22	0.01	4.72
9	59,050	314,955	20	36	410	0.83	0.79	41.11
10	196,831	1,181,010	22	40	572	4.56	3.54	398.5
11	649,540	4,330,282	24	44	772	35.93	34.08	$\gg^2$

TABLE 5.2 – Vérification distribuée de formules CTL pour le problème des gestionnaire des bases de bases de données réparties

Pour chacun des deux problèmes traités, le temps d'exécution, le nombre de messages, ainsi que le nombre de contre-exemples dépendent de la taille de l'espace d'états et de la taille de la formule CTL vérifiée, et en particulier du nombre d'opérateurs temporels dans la formule.

5.6 Conclusion

Dans ce chapitre, nous avons proposé une démarche simple pour distribuer le modèle checking, dans le cas de la logique temporelle CTL, sur un espace d'états distribué. Cela permet de verifier des systèmes complexes et de taille importante tout en adoptant une représentation explicite des états. D'autre part, une recherche du contre-exemple est menée dans le cas où la formule vérifiée n'est pas valide, et ce en se basant toujours sur l'espace d'états réparti. Plusieurs tests ont été menés sur un cluster et une accélération super linéaire est obtenue dans certains cas.

Chapitre 6

Analyse distribuée des systèmes modulaires

6.1 Introduction

L'approche basée sur l'analyse modulaire permet la vérification de manière isolée des sous-systèmes (modules) qui composent un système donné.

Dans cette partie, on se propose d'utiliser cette structure modulaire afin de réaliser une vérification distribuée basée sur une répartition des différents modules sur un ensemble de processus.

Dans la première étape, chacun des processus génère le graphe des marquages local et contribue à la construction du graphe de synchronisation. Ensuite, il permet la vérification de propriétés générales telles que l'atteignabilité, la vivacité, l'existence d'états de bloquage.

6.2 Les Réseaux de Petri Modulaires

Nous considérons les réseaux de Petri modulaires avec partage de transitions tels que définis dans [18].

Définition 22 *Réseaux de Petri Modulaires* ***Un réseau de Petri modulaire*** *est le couple* $MN = (S, TF)$ *tel que :*

1. *S est un ensemble fini de modules tel que :*

- *Chaque module $s \in S$ est un réseau de Petri : $s = (P_s, T_s, W_s, M_{s_0})$.*
- *L'ensemble de nœuds des différents modules sont deux à deux disjoints :* $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.
- $P = \bigcup_{s \in S} P_s$ *et* $T = \bigcup_{s \in S} T_s$ *constituent respectivement l'ensemble des places et l'ensemble des transitions de tous les modules du système.*

2. $TF \subseteq 2^T$ *est un ensemble fini non vide d'**ensembles de transitions de fusion**.*

Dans ce qui suit TF désignera aussi $\bigcup_{tf \in TF} tf$.

Définition 23 *Groupe de transitions Un **groupe de transitions** $tg \subseteq T$ consiste soit en une transition qui n'est pas une transition de fusion $t \in T \setminus TF$, soit en l'ensemble des élements d'un ensemble de fusion de transitions $tf \in TF$.*
L'ensemble des groupes de transitions est noté TG.

Une transition peut être élément de plusieurs groupes de transitions puisqu'elle peut être synchronisée avec différentes transitions. Ainsi, un groupe de transitions correspond à une action synchronisée.

La fonction de poids des arcs W est étendue aux groupes de transitions :

$$\forall p \in P, \forall tg \in TG, \qquad W(p, tg) = \sum_{t \in tg} W(p, t), \quad W(tg, p) = \sum_{t \in tg} W(t, p)$$

Le marquage d'un réseau de Petri modulaire est défini de la même manière que dans les réseaux de Petri ordinaires, sur l'ensemble des places de tous les modules. La restriction d'un marquage M à un module s est notée M_s. Les règles de sensibilisation et de franchissement des transitions d'un réseau de Petri modulaire peuvent alors être exprimées.

Définition 24 *Franchissement d'un groupe de transitions Un groupe de transitions tg est sensibilisé pour un marquage M, noté par $M[tg\rangle$, si et seulement si :*

$$\forall p \in P : W(p, tg) \leq M(p)$$

Lorsqu'un groupe de transitions tg est sensibilisé pour un marquage M_1, il peut être franchi, donnant le nouveau marquage M_2, défini par :

$$\forall p \in P : M_2(p) = (M_1(p) - W(p, tg)) + W(tg, p)$$

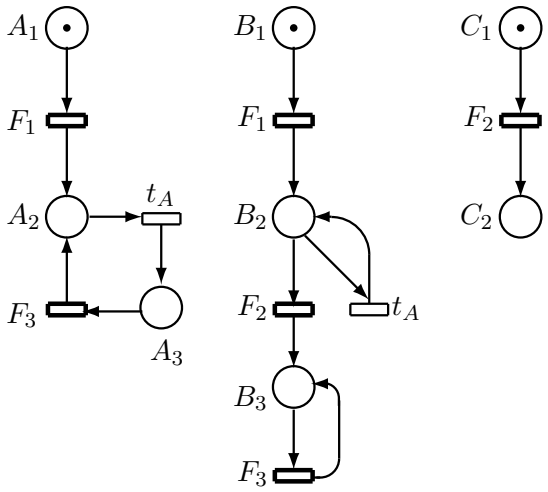

FIGURE 6.1 – Un réseau de Petri modulaire avec 3 modules

La figure 6.1 illustre un réseau de Petri modulaire qui consiste en trois modules A, B et C. Les modules A et B comportent tous les deux les transitions étiquetées F_1 et F_3, tandis que B et C contiennent la transition F_2. Ces transitions sont alors considérées comme étant des transitions de fusion et forment l'ensemble des transitions de fusion.

6.3 Espace d'états modulaire

Nous désignons par $[[M\rangle$ l'ensemble des états atteignables à partir du marquage M par le franchissement de transitions internes seulement (pas de transition de fusion).

La notation avec l'indice s désigne la restriction au module s, c'est-à-dire $[M\rangle_s$ est l'ensemble des marquages atteignables à partir du marquage global M en franchissant seulement les transitions du module s.

On utilise $M_1[[\sigma\rangle\rangle M_2$ pour désigner le fait que M_2 est atteignable à partir de M_1 par le franchissement de la séquence $\sigma \in T \setminus TF)^*TF$ constituée de transitions internes suivies d'une transition de fusion.

Pour tout marquage accessible M, on utilise $M^{\mathscr{q}}$ pour désigner le produit (ou le tuple) des composantes fortement connexes M_s^c des différents modules :

$$\forall M \in [M_0\rangle : M^{\mathscr{q}} = \prod M_s^{\mathscr{q}}$$

L'espace d'états modulaire consiste en deux parties : les espaces d'états des différents modules et le graphe de synchronisation.

Définition 25 *Soit $MN = (S, TF)$, un réseau de Petri modulaire avec le marquage initial M_0. L'**espace d'états modulaire** de MN est un couple $MSS = ((SS_s)_{s \in S}, SG)$ où :*

1. *$SS_s = (V_s, A_s)$ est l'espace d'états local du module s :*
   (a) $V_s = \bigcup_{v \in V_{SG}} [v\rangle_s$
   (b) $A_s = \{(M_1, t, M_2) \in V_s \times (T \setminus TF)_s \times V_s \mid M_1[t\rangle M_2\}$
2. *$SG = (V_{SG}, A_{SG})$ est le **graphe de synchronisation** de MN :*
   (a) $V_{SG} = [[M_0\rangle\rangle^c \cup M_0^c$.
   (b) $A_{SG} = \{(M_1^c, (M_1'^c, tf), M_2^c) \in V_{SG} \times ([M_0\rangle^c \times TF) \times V_{SG} \mid M_1' \in [[M_1\rangle \wedge M_1'[tf\rangle M_2\}$

L'ensemble des nœuds de l'espace d'états d'un module (1.a) contient tous les états atteignables à partir de n'importe quel nœud du graphe de synchronisation.

Les arcs du graphe de l'espace d'états d'un module (1.b) correspondent à toutes les transitions internes franchissables du module.

Chaque nœud du graphe de synchronisation est étiqueté par $M^{\mathcal{G}}$ et il est représentatif de tous les nœuds atteignables à partir de M en franchissant seulement des transitions internes, c'est-à-dire $[[M\rangle\rangle$. Le graphe de synchronisation comporte les informations sur les nœuds atteignables lors du franchissement des transitions de fusion. Les graphes des espaces d'états des modules comportent donc des informations locales, c'est-à-dire les marquages des modules et les arcs correspondants aux transitions internes mais pas les transitions de fusion.

Les nœuds du graphe de synchronisation (2.a) représentent tous les marquages accessibles à partir d'autres marquages en franchissant une séquence de transitions internes suivie d'une transition de fusion. Le marquage initial est aussi représenté.

Les arcs du graphe de synchronisation représentent les occurrences des transitions de fusion. Les étiquettes des arcs du graphe de synchronisation comportent le nom de la transition franchie, ainsi que sa source. Cette dernière correspond au produit des composantes fortement connexes auxquelles appartiennent les états qui sensibilisent cette transition.

Exemple 1

L'espace d'états modulaire du réseau de Petri modulaire de la figure 6.1 est donné dans la figure 6.2

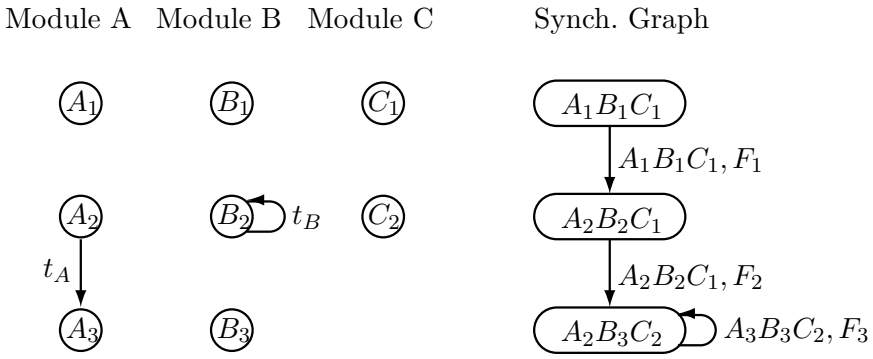

FIGURE 6.2 – L'espace d'états modulaire de l'exemple de la fig. 6.1

6.4 Construction de l'espace d'états d'un réseau de Petri modulaire

Plusieurs outils ont été developpés pour permettre la génération et la vérification distribuées de l'espace d'états sur des clusters. L'espace d'état est partitionné sur les différentes machines d'un cluster, utilisant des fonctions de hashage pour déterminer la station qui sauvegardera chaque état. Cette approche permet de traiter des problèmes de taille plus grande, mais reste néanmoins limitée.

Dans ce chapitre, nous considérons une nouvelle approche basée sur la modularité pour la construction distribuée de l'espace d'états ainsi que la vérification de diverses propriétés.

Deux types de processus seront utilisés. Ils consistent en un **processus maître** et **N processus esclaves**.

Le processus maître se charge de la construction du graphe de synchronisation, coordonne les processus esclaves et détermine la fin de la construction de l'espace d'états ou de la vérification. Tandis que chaque processus esclave s'occupe de la génération de l'espace d'états local, du module qui lui est assigné, par le franchissement de transitions internes seulement.

L'algorithme consiste en trois étapes : dans la première étape, l'espace d'états potentiel local de chaque module est construit par le processus auquel il a été affecté en considérant que chaque module est indépendant des autres ; dans la seconde phase, le graphe de synchronisation est construit par le processus maître (en ne considérant que les transitions de synchronisation réellement franchissables) ; et enfin, dans la troisième étape, les états locaux non atteignables sont supprimés.

Chaque processus esclave s commence la génération de l'espace d'états à partir de l'état initial $M_0(s)$ en franchissant les transitions internes. Il envoie les marquages qui sensibilisent des transitions de synchronisation au processus maître.

L'algorithme 14, génére l'ensemble des états locaux de manière itérative. Chaque itération est composée de deux étapes : dans la première étape, on génère tous les marquages possibles par le franchissement de transitions internes. Les marquages qui sensibilisent des transitions de fusion sont stockés, par chaque processus s, dans $Synch_s$. Lorsqu'aucune transition interne n'est sensibilisée, on choisit dans une deuxième étape, un des marquages de $Synch_s$ puis on tire la ou les transitions de synchronisation qu'il sensibilise. Si le marquage généré permet le franchissement de transitions internes, on reprend alors une nouvelle itération.

Dans la deuxième étape des messages concernant les transitions de fusion sont envoyés au processus `Maître`. Ces messages contiennent la composante fortement connexe M_s^c du marquage M_s permettant le franchissement d'une transition de fusion tf, la transition tf, le numéro de la composante fortement connexe du marquage généré $M_s^{'c}$ par le tir de cette transition ainsi que les numéros des composantes fortement connexes de ses prédécesseurs dans le graphe de synchronisation $AncSCC(M_s)$ (appelés Ancêtres).

La fonction `AddNode` (M_s) rajoute le marquage M_s à l'ensemble des marquages locaux V_s et à $Waiting_s$ s'il n'était pas déjà dans V_s.

Les communications étant asynchrones, à la réception d'un message, le processus est interrompu et la fonction de traitement des messages `MessageHandler`() est invoquée. L'algorithme 15 décrit les actions réalisées par le processus maître lorsque celui-ci reçoit un message comportant un marquage sensibilisant une transition de fusion du processus esclave s.

En premier, le processus `Maître` crée le nœud initial du graphe de synchronisation $v_0 = M_0^c \in V_{SG}$ correspondant aux composantes fortement connexes des marquages initiaux des différents modules. Lorsqu'une transition de synchronisation tf est reçue du processus s les successeurs du nœud $v = (v_1, v_2, \ldots v_N) \in V_{SG}$ sont construits de la manière suivante :

Procedure 14 Internal State Space Generation()

```
begin
    AddNode (M_0(s))
    repeat
        /* Process states in Waiting_s */
        while Waiting_s ≠ ∅ do
            Choose M_s ∈ Waiting_s
            foreach t ∈ T_s such that M_s[t⟩M'_s do
                if t ∈ TF then
                    /* M_s enables a fused transition */
                    Synch_s = Synch_s ∪ {M_s};
                else
                    /* t is an internal transition */
                    AddNode (M'_s)
                    AddArc (M_s, t, M'_s)
        SCCUpdate()
           /* Process marking wich enable fused transitions */
        if Synch_s ≠ ∅ then
            Choose M_s ∈ Waiting_s
            foreach tf ∈ TF such that M_s[tf⟩M'_s do
                /* M_s enables a fused transition */
                AddNode (M'_s)
                Send (Coordinator, (M^c_s, tf, M'^c_s, AncSCC(M_s))
    until Waiting_s = ∅ ∧ Synch_s = ∅
    Send (Coordinator, END_GENERATION)
end
```

- On considère l'ensemble PM des modules (processus) participants à la synchronisation de la transition de fusion tf ;
- On construit les ensembles :
  - W_{pm}, correspondant aux transitions de fusion reçues des différents processus `Esclaves`.
  - W_s, correspondant aux transitions de fusion reçues du processus `Esclave` s.
  - W_{np}, qui contient $\bullet$ pour les modules ne participant pas la synchronisation.
- pour toutes les combinaisons, les successeurs dans le graphe de synchronisation sont déterminés et insérés.

Procedure 15 Message Handler()

begin

⋮

if $Message.Type =$ `SynchTrans` **then**

`/* message received contains an enabled fused transition */`

$tf = Message.Content.tr$

$s = Message.Sender$

$PM = \{i \text{ s.t. } tf \in T_i\}$

`/* PM: modules participating in the synchronisation of` tf `*/`

foreach $v = (v_1^c, v_2^c, \cdots, v_N^c) \in V_{SG}$ **do**

foreach $pm \in PM$ *and* $pm \neq s$ **do**

$W_{pm} = \{w \in SGwaiting_{pm} \text{ s.t. } (w.AncSCC = v_{pm}^c) \wedge (w.tr = tf)\}$

`/*` W_{pm}`: markings enabling` tf`, with` v_{pm} `as ancestor */`

foreach $np \notin PM$ *and* $np \neq s$ **do**

$W_{np} = \{\bullet\}$

`/* any marking of a non-participating module enables synchronisation */`

$W_s = \{Message.Content\}$

foreach $C = (c_1, c_2, \cdots, c_N)$ *s.t.* $\forall i \in PM, c_i \in W_i$ **do**

$v' = (v'^c_1, v'^c_2, \cdots, v'^c_N)$ where :

$\forall i \in PM, v'^c_i = c_i.SuccSCC$ and $\forall i \notin PM, v'^c_i = v_i^c$

`add` (v')

`AddArc` $(v, (C, ft), v')$

foreach $pm \in PM$ **do**

`Send` $(Sender = Coordinator, Dest = pm, Type =$ `NEW_NODE`$,$ $Content = v'_{pm})$

$SGwaiting_s = SGwaiting_s \cup Message.Content$

⋮

end

Les nœuds du graphe de synchronisation sont envoyés aux processus `Esclaves` (avec une projection sur le processus concerné) pour qu'ils procèdent à la suppression des états non atteignables.

6.5 Vérification des propriétés

Le graphe des marquages est le modèle de base utilisé pour la vérification de la majorité des propriétés. Dans ce qui suit, nous nous concentrons sur la vérification distribuée de propriétés comportementales telles que : atteignabilité, états de blocage, vivacité et états d'accueil dans les systèmes modulaires.

6.5.1 L'atteignabilité

L'atteignabilité est une propriété décidable. Dans notre cas, nous proposons d'effectuer la vérification de cette propriété de manière distribuée et sur des systèmes modulaires. Ainsi, pour vérifier si un marquage M est atteignable, chaque processus *esclave* s recherche dans son espace d'états local V_s si M_s, la restriction de M au module s, existe ou pas. Si M_s n'existe pas, on peut conclure que le marquage M n'est pas atteignable, sinon chaque processus *esclave* s envoie au processus *Maître* les numéros des CFC à partir desquelles le marquage M_s est atteignable.

Le Maître enregistre dans W_s les composantes fortement connexes reçues du processus s. Puis il vérifie s'il existe une combinaison $C = (c_1, c_2, \ldots, c_N)$ telle que $c_i \in W_i$ appartenant au graphe de synchronisation. C'est dans ce cas seulement, qu'on peut conclure que le marquage M est atteignable.

6.5.2 Etats bloquants

L'algorithme utilisé pour trouver les états bloquants est basé sur la proposition suivante :

Proposition 1 *États puits* $M \in [M_0\rangle$ *est un état puits* $\Leftrightarrow [\forall s \in S : (Ms)^c \in Term(SCC_s) \cap Trivial(SCC_s)) \wedge (\forall (v_1, (M_1^c, tf), v_2) \in A_{SG} : M_1^c \neq M^c)]$

Chaque processus *esclave* s détermine les marquages puits dans son espace local. Si un des processus ne possède pas d'états bloquants, le système est alors sans états bloquants. Si

tous les processus *esclave* trouvent des états puits, ils les transmettent au processus *Maître* (ce sont aussi les numéros des composantes fortement connexes qui sont transmis). Le processus *Maître* sauvegarde dans D_s les composantes fortement connexes reçues du processus *esclave* s, puis vérifie pour chaque combinaison si elle n'est pas parmi les étiquettes des transitions de fusion. Dans ce cas, le marquage correspondant n'est pas un état de blocage. Sinon, on vérifie si le marquage est atteignable, et si c'est le cas alors il est bloquant.

6.5.3 La vivacité

On considère ici deux cas : le premier cas concerne les transitions de fusion et le second, les transitions internes. La vérification est basée sur la proposition suivante :

Proposition 2 *Vivacité*

1. *Une transition* $tf \in TF$ *est vivante* $\Leftrightarrow [\forall scc \in Term(SCC_{SG} : tf \in Trans(scc)]$
   $\wedge[\forall v \in V_{SG} : \forall M \in [[v\rangle : (\forall s \in S : M_s^c \in Term(SCC_s)) \Rightarrow \exists (v, (M_1, tf'), v_2) \in A_{SG} :$
   $M_1 \in [[M\rangle]$.
2. *La transition* $t \in T_s$ *est vivante* $\Leftrightarrow$
   $[\forall scc \in Term(SCC_{SG} : \exists v \in scc : t \in Trans([v_s\rangle_s)]$
   $\wedge[\forall v \in V_{SG} : \forall M \in [[v\rangle : (M_s^c \in Term(SCC_s)) \Rightarrow (t \in Trans(M_s^c) \vee \exists (v, (M_1, tf), v_2) \in$
   $A_{SG} : M_1 \in [[M\rangle))]$.

Pour vérifier si une transition de fusion tf est vivante, le processus Maître vérifie s'il existe une CFC terminale qui ne comporte pas la transition tf, si tel est le cas la transition est alors non vivante. Sinon, pour chaque nœud $v \in V_{SG}$ du graphe de synchronisation, le processus Maître envoie v_s au processus s et attend de recevoir les composantes fortement connexes terminales atteignables à partir de v_s dans le module s, puis vérifie si l'une des combinaisons n'est pas une étiquette d'une transition de fusion. Dans ce cas aussi la transition tf est non vivante.

Lorsque la transition t est interne et appartient au module s, le processus s détecte les composantes fortement connexes terminales où t n'est pas franchissable, ces CFC sont alors dites *problématiques*. De la même manière, le processus Maître envoie chaque nœud v du graphe de synchronisation aux processus esclaves, pour lui réenvoyer les composantes fortement connexes problématiques atteignables à partir de v. Le Maître vérifie alors s'il y a des combinaisons qui ne sont pas des étiquettes de transitions de synchronisation et dans ce cas la transition t est non vivante.

6.5.4 Etat d'accueil

La vérification qu'un état M_H est un état d'accueil est basée sur la proposition suivante :

Proposition 3 *Un état $M_H \in [M_0\rangle$ est un état d'accueil si et seulement si :*
$[\forall scc \in Term(SCC_{SG} : \exists v \in scc : M_H \in [[v\rangle]$
$\wedge[\forall v \in V_{SG} : \forall M \in [[v\rangle : (\forall s \in S : M_s^c \in Term(SCC_s))$
$\Rightarrow M_H \in [[M\rangle \vee \exists(v, (M_1, tf, M_2), v_2) \in A_{SG} : M_1 \in [[M\rangle.$

On vérifie tout d'abord que M_H est atteignable à partir de toutes les composantes fortement connexes terminales du graphe de synchronisation. Puis, pour un nœud v du graphe de synchronisation, le Maître demande au processus esclave de lui envoyer toutes les composantes fortement connexes terminales atteignables à partir de v et ne contenant pas M_H. Le Maître vérifie alors que toutes les combinaisons sont des étiquettes de transitions de synchronisation, dans le cas contraire M_H n'est pas un état d'accueil.

6.6 Implémentation et tests

Les algorithmes précédents ont tous été implémentés dans un prototype. Des tests ont été menés sur un cluster composé de 12 stations, une des stations est utilisée pour le Maître, tandis que les autres sont utilisées pour les différents processus esclaves.

Dans cette partie nous donnons les résultats obtenus pour les tests effectués sur *le problème des philosophes* et le *problème des véhicules auto-guidés(AGV).*

6.6.1 Problème des philosophes

Dans l'approche de distribution basée sur la modularité, le réseau de Petri a été découpé en N modules, et chacun des modules, contenant m philosophes, est assigné à un processus `esclave`. Le philosophe i du module j partage sa fourchette avec les philosophes $i-1$ et $i+1$ du même module lorsque $2 \leq i \leq m-1$. Le philosophe 1 du module l et le philosophe m dans le module $(l-1) mod N$ partagent aussi leurs fourchettes.

Si nous considérons, par exemple, 10 philosophes par module, nous aurons 233 nœuds et 1132 arcs dans les graphes internes, et 2^N nœuds et $N.2^N$ arcs dans le graphe de synchronisation. Ainsi, pour le problème de 100 philosophes, on a 1024 nœuds dans le graphe de synchronisation et 233 nœuds dans chaque graphe interne. Le graphe des marquages

ordinaire correspondant au problème de 100 philosophes comporte plus que 7×10^{20} nœuds et plus de 7×10^{22} arcs.

Dans la Table 6.1, nous donnons les temps CPU moyens pour la génération de l'espace d'états en distribué ainsi que le temps CPU correspondant au cas séquentiel.

Nb Philo	Nodes N	Arcs	Mess Nb	CPU Time (sec)	CPU time (1 proc)
20	470	2162	26	0.04	0.08
40	948	4372	52	0.04	0.16
60	1462	6846	84	0.04	0.25
80	2120	10664	120	0.05	0.39
100	3354	21010	230	0.16	0.61

TABLE 6.1 – Génération modulaire distribuée

Des tests concernant l'atteignabilité, la vivacité et la propriété d'état d'accueil ont été realisés avec le problème des philosophes (10 modules de 10 philosophes). Les résultats sont donnés dans la table 6.2.

Properties	CPU Time (sec)	Mess Nb
Reachability1	< 0.01	20
Reachability2	0.03	20
Reachability3	0.02	20
Liveness ($Internal\ tr$)	0.17	4278
Liveness ($Fuzed\ tr$)	0.15	4116
Home state	0.21	4432

TABLE 6.2 – Vérification modulaire distribuée de propriétés (exemple des philosophes)

Pour les tests d'atteignabilité, nous avons considéré deux cas : le premier consiste à vérifier l'atteignabilité d'un marquage atteignable dans chaque module mais non atteignable de manière globale ; dans le second nous avons considéré un marquage atteignable.

6.6.2 Problème des véhicules auto-guidés

Le problème des véhicules auto-guidés a été resolu en utilisant la modularité dans [18]. Le problème consiste en une usine constituée de trois postes de travail qui opérent sur des pièces, deux postes d'entrée et un de sortie, avec 5 véhicules auto-guidés qui transportent des pièces d'un poste à un autre. Ce problème est faiblement couplé, ainsi beaucoup d'entrelacement sont évités lors de la construction de l'espace d'états modulaire et ceci permet

d'obtenir de très bons résultats. Ce modèle contient à l'origine 30,965,760 états. Cependant, avec la construction modulaire on obtient seulement 900 états avec 2687 arcs. L'analyse dans le cas distribué donne aussi de bons résultats comme on peut le voir dans le tableau 6.3.

	CPU Time (sec)	Mess Nb
State Space Generation	0.02	82
Deadlocks search	0.03	11
Reachability1	< 0.01	22
Reachability2	0.02	22
Reachability3	0.02	22
Liveness ($Internal\ tr$)	0.09	3313
Liveness ($Fuzed\ tr$)	0.08	3062
Home state	0.15	3415

TABLE 6.3 – Vérification de propriétés distribuées pour le problème des 5 AGVs

6.7 Conclusion

Dans ce chapitre, nous avons proposé les étapes nécessaires pour l'analyse distribuée modulaire des réseaux de Petri. basées sur le travail effectué dans [18]. L'utilisation des algorithmes de vérification modulaire permet l'analyse de propriétés standard sur les réseaux de Petri, telles que l'atterignabilité, le blocage, la vivacité et l'état d'accueil, de manière distribuée.

L'avantage d'une telle approche est la possibilité d'analyser des systèmes de très grande taille avec de nombreux modules, ayant des transitions partagées, appelées transitions de fusion ou transitions de synchronisation. Les modules peuvent être assignés à un ensemble de machines reliées en cluster.

Chaque machine (processus) génère seulement l'espace d'états du module qui lui a été affecté. Le graphe de synchronisation, généré par le processus `maître`, sert à connaître le comportement global du système.

Comme dans le cas séquentiel, l'analyse modulaire est particulièrement adaptée pour les systèmes avec une forte cohésion et un couplage faible. Ces caractéristiques sont toujours souhaitables dans les systèmes modulaires. Lorsqu'un système présente un fort couplage entre sous-systèmes, les avantages de l'analyse modulaire sont moins marqués.

Des expriences ont été réalises sur un cluster de 12 stations, Le problème des véhicules auto guidés (AGV) et le problème des philosophes, avec des tailles différentes ont été en-

visagés. Les résultats obtenus pour la génération de l'espace d'états modulaire distribué étaient très intéressants et permettent de vérifier des propriétés pour des systèmes complexes de tr‘es grande taille.

Conclusion Générale

Partant de la nécessité de prouver la correction des systèmes critiques, plusieurs travaux se sont consacrés à la construction d'outils d'analyse et de vérification de ces systèmes. Les méthodes de vérification basées sur le model-checking permettent de prouver de manière formelle qu'un système satisfait ou non les bonnes propriétés attendues. Cette approche de vérification repose sur la génération de l'espace d'états, elle a l'avantage d'être complètement automatique et elle offre une analyse fine et exhaustive. Cependant, le problème de l'explosion combinatoire de l'espace d'états constitue une vraie limite de cette approche. Pour pouvoir vérifier des systèmes de taille importante, nous nous sommes intéressés à la vérification distribuée.

L'idée consiste à utiliser un ensemble de machines, connectées en réseau et communiquant par envois de messages, offrant ainsi un plus grand espace mémoire et une puissance de calcul accrue. Il s'agit alors de partitionner l'espace d'états sur ces machines et de les faire coopérer afin d'effectuer des vérifications de propriétés sur cette espace d'états réparti.

L'algorithme de génération de l'espace d'états est exécuté en parallèle. Chaque machine génère une partie de l'espace d'états. Une fonction de répartition (hashage) est utilisée afin de déterminer le processus propriétaire de chaque état. Le choix de la fonction de hashage est déterminant pour les performances des algorithmes de vérification. En effet, cette fonction doit assurer un équilibre de charge entre les différents processus, minimiser les arcs traversants pour éviter un nombre trop grand d'échanges de messages.

La construction d'une bonne fonction de hashage n'est pas aisée, et dépend de la nature du problème traité. Nous avons décrit les différents types de fonctions, statiques et dynamiques, et proposé une répartition basée sur la virtualisation de processeurs [56]. Cette méthode utilisant les possibilités offertes dans charm++ et MPI, permet une affectation dynamique des processus et la migration de processus d'un processeur chargé vers un autre moins chargé. Cela nous a permis d'obtenir un bon équilibre de charge entre les différentes

stations. et peut permettre de concentrer les efforts sur la minimisation du nombre d'arcs traversants.

Nous avons aussi proposé des algorithmes de vérification de propriétés dites générales telles que l'atteignabilité, l'existence de blocage, la vivacité et l'état d'accueil [11]. Les deux premières propriétés peuvent être vérifiées par un simple examen des états ou pendant la génération (à la volée) et permettent d'obtenir des accélérations supra-linéaires. La vérification des propriétés de vivacité et d'état d'accueil nécessite la construction de composantes fortement connexes. Un algorithme basé sur le marquage des ascendants et des descendants a été adapté au cas distribué afin de déterminer les composantes fortement connexes du graphe des marquages réparti.

La vérification de propriétés spécifiques exprimées dans une logique temporelle a aussi fait l'objet de notre étude. Nous avons décrit les difficultés rencontrées dans le Model-checking distribué pour la logique temporelle LTL. Les algorithmes de vérification étant basés sur un double parcours en profondeur (Nested Depth First Search NDFS), ce type de parcours n'est pas bien adapté pour le calcul distribué. Nous avons proposé un algorithme basé sur la contraction en parallèle des composantes fortement connexes des sous-graphes internes de chaque processus, puis de construire à partir du graphe réduit un graphe de dépendances.

Des algorithmes distribués pour la vérification de formules CTL, sur un espace d'états réparti, ont été proposés. Un contre-exemple est déterminé dans le cas où la formule CTL n'est pas satisfaite [13].

Une autre approche de vérification distribuée basée sur la modularité a été aussi proposée [12]. Dans cette approche, chaque machine (processus) génère seulement l'espace d'états du module qui lui a été affecté. Le graphe de synchronisation, généré par le processus `maître`, sert à connaître le comportement global du système. L'avantage d'une telle approche est la possibilité d'analyser des systèmes de très grande taille avec de nombreux modules, ayant des transitions partagées, appelées transitions de fusion ou transitions de synchronisation. Les modules peuvent être assignés à un ensemble de machines reliées en cluster. Comme dans le cas séquentiel, l'analyse modulaire est particulièrement adaptée pour les systèmes avec une forte cohésion et un couplage faible. Ces caractéristiques sont toujours souhaitables dans les systèmes modulaires. Lorsqu'un système présente un fort couplage entre sous-systèmes, les avantages de l'analyse modulaire sont moins marqués.

La conjugaison de l'approche de l'analyse modulaire et de l'approche distribuée ont permis d'obtenir de très bons résultats. Ainsi, pour le problème des philosophes par exem-

ple, nous avons pu générer l'espace d'états d'un problème dont la taille normale est de plus de 6.10^{23} états.

La majorité des algorithmes proposés dans cette thèse ont fait l'objet d'un prototype pour effectuer des tests. Les résultats obtenus sur des problèmes classiques, tels que le problèmes des philosophes, le problèmes des gestionnaire de bases de données réparties et les problèmes des véhicules auto-guidés, sont encourageants.

Perspectives La vérification distribuée est une autre approche qui permet de juguler le problème de l'explosion combinatoire. Néanmoins, beaucoup de questions restent soulevées ou à améliorer, en particulier :

- programmation basée sur des processeurs multi-cœurs et la virtualisation en profitant des nouvelles technologies de processeurs afin de réduire le temps de l'analyse des systèmes.
- Implémenter la méthode de réduction basée sur la contraction des composantes fortement connexes, pour minimiser la taille du graphe de dépendances dans la vérification distribuée des formules LTL.
- Il y a aussi des travaux qui s'intéressent à la distribution d'autres approches de vérification, telle que la vérification ou la réduction de l'espace d'états basée sur la bisimulation. Cette voie reste encore à explorer.
- Dans la vérification modulaire distribuée, la génération du graphe de synchronisation est prise en charge par un seul processus, ce qui limite la montée en échelle et la taille des problèmes traités, particulièrement lorsqu'ils sont fortement couplés, on peut penser alors à la génération distribuée de ce graphe.

Bibliographie

[1] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, aug 1996.

[2] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability anlysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems : Models and Application*, volume 103 of *LNCIS*, pages 40–56. Springer, 1987.

[3] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear temporal model checking : One step beyond. In Springer, editor, *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, 1998.

[4] E. Allen Emerson and Edmund M. Clarke. *Characterizing correctness properties of parallel programs using fixpoints*. Springer, 1980.

[5] Soheib Baarir. *Exploitation des symétries partielles pour la vérification et l'évaluation de performances des systèmes finis*. PhD thesis, Paris VI, 2007.

[6] J Barnat, L Brim, and J St Vr Ibrná. Distributed ltl model-checking in spin. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 200–216, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[8] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L.Petrucci, and Ph.Schnoebelen. *Systems and Software Verification : model-checking techniques and tools*. Springer Verlag, 2001.

[9] B. Berthomieu and M. Menashe. An enumerative approach for analysing time petri nets. In R.E.A. Mason, editor, *Information processing : Proceeding of the the IFIP congress*, volume 9 of *Elsevier Scince Publishers, Amsterdam*, pages 41–46, 1983.

[10] S. Blom and S. Orzan. Distributed state space minimization. *Electronic Notes in Theoretical Computer Science*, 80 :1–15, 2003.

[11] M.C. Boukala and L. Petrucci. Towards distributed verification of Petri nets properties. In *Proc. 1st International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS'07), Algiers, Algeria*, eWiC, pages 15–26. British Computer Society, May 2007.

[12] M.C. Boukala and L. Petrucci. Distributed verification of modular systems. In *In Proc. workshop on Petri Nets Compositions (CompoNet'11, associated with Petri Nets 2011)*, Volume 726 of CEUR workshop proceedings, pages 1–15, June 2011.

[13] M.C. Boukala and L. Petrucci. Distributed model-checking and counterexample search for CTL logic. *International Journal of Critical Computer-Based Systems, IJCCBS*, 3(1-2) :44–59, January 2012.

[14] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Proc. Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2001.

[15] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293 318, September 1992.

[16] Stanislas Budkowski and Piotr Dembinski. An introduction to ESTELLE : A specification language for distributed systems. *Computer Networks and ISDN Systems*, 1988.

[17] Annie Choquet-Geniet. *Les réseaux de Petri - Un outil de modélisation.* Dunod, 2006.

[18] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3) :224–242, 2000.

[19] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 1998.

[20] Gianfranco Ciardo. Distributed and structured analysis approaches to study large and complex systems. In *Lectures on Formal Methods and Performance Analysis, LNCS 2090*, pages 344–374. Springer-Verlag, 2001.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages p. 244–263, April 1986.

[22] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Int. Conf. Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer, 1993.

[23] E. M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. *Proceedings : IEEE Symposium on Logic in Computer Science 2002 (LICS '02)*, 2002.

[24] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. in computer-aided verification '90 (new brunswick, nj, 1990). DIMACS Ser, vol 3 :p. 207–218,Springer–Verlag, 1991.

[25] Alan J. Hu David L. Dill, Andreas J. Drexler and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design : VLSI in Computers and Processors, IEEE Computer Society)*, pages 522–525. IEEE Computer Society, 1992.

[26] M. Diaz. *Les réseaux de Petri, modèles fondamentaux*. Hermes Science publication, 2001.

[27] M. Dwyer, G. Avruin, J. Corbett, and Y. Hu. Patterns in property specification for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. In M. Ardis, editor, 1977.

[28] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[29] Sami Evangelista, Laure Petrucci, and Samir Youcef. Parallel nested depth-first searches for ltl model checking. In *Proceedings of the 9th international conference on Automated technology for verification and analysis*, ATVA'11, pages 381–396, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] Brams G, W. *Réseaux de Petri. Théorie et pratique*. Masson, 1983.

[31] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer, 2001.

[32] Brent Hailpern and Susan Owicki. Modular verification of concurrent programs. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–336, New York, NY, USA, 1982. ACM.

[33] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–585, 1969.

[35] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[36] G. Holzmann. The spin model-checker. *IEEE Transactions on Software Engineering*, vol. 23, no 5 :279–295, 1997.

[37] Gerard J. Holzmann. *SPIN Model Checker, The : Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[38] C. P. Inggs and H. Barringer. Ctl* model checking on a shared-memory architecture. *Formal Methods in System Design*, Department of Computer Science, 2005.

[39] K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modeling and Validation of Concurrent Systems*. Springer-Verlag Berlin, 2009.

[40] Christophe Joubert. *Vérification distribuée à la volée de grands espaces d'états*. PhD thesis, Institut National Polytechnique de Grenoble, 2005.

[41] Laxmikant V. Kale. Performance and productivity in parallel programming via processor virtualization. Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10), February 2004.

[42] L.V. Kale and S. Krishnan. *Charm++ : Parallel Programming with Message-Driven Objects. In ?Parallel Programming using C++ ?* MIT Press, 1996.

[43] L. Kristensen and L. Petrucci. An approach to distributed state exploration for coloured Petri nets. In *Proc. 25th Int. Application and Theory of Petri Nets (ICATPN'2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 474–483. Springer, 2004.

[44] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. *Electr. Notes Theor. Comput. Sci*, 128, 2005.

[45] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *Proc. 5th and 6th International SPIN workshops on Model Checking Software (SPIN'1999)*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1999.

[46] Fujita M, Tanaka H, and Moto-oka T. Logic design assistance with temporal logic. In *Proceedings of the Seventh International Symposium on Computer Hardware Description Languages and Their Applications*, pages 129–138. IFIP, 1985.

[47] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. Springer Verlag, 1991.

[48] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[49] K. McMillan. An introduction to model-checking. *Actes de l'école d'été MOVEP'98*, juillet 6th-9th 1998.

[50] Robin Milner. A calculus of communicating systems. *LNCS 92*, 1980.

[51] Robin Milner. Communication and concurrency. *International Series in Computer Science*, 1989.

[52] T. Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[53] R. Nalumasu and G. Gopalakrishnan. A new partial order reduction algorithm for concurrent systems. In *Proc. Int. Conf. Hardware Description Languages and their Applications (CHDL'97)*. Chapman & Hall, 1997.

[54] David Park. Concurrency and automata on infinite sequences. In Springer, editor, *In Deussen, P. (ed.). Theoretical Computer Science, Proceedings of the 5th GI-Conference Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, 1994.

[55] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[56] Messaoud Rahim and Mohand Cherif Boukala. Using processor virtualization to load balancing distributed state space construction. In *Proceedings of the 10th International Symposium on Programming ans Systems*, pages 156–162, 2011.

[57] K. A. Robinson. The B-method and the B-toolkit. In *Sixth International Conference, Algebraic Methodology and Software Technology*, pages 576–580. Sydney, Springer, 1997.

[58] Rodrigo Saad, Bernard Berthomieu, Silvano Dal Zilio, and François Vernadat. Enumerative parallel and distributed state space construction. In *ETR09 - École d'été Temps Réel*, Paris France, 2009. French AESE project Topcased.

[59] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking.* Vuibert, APR 1999.

[60] Frank L. Severance. *System Modeling and Simulation : An Introduction.* John Wiley and sons, 2001.

[61] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7 :67–71, 1981.

[62] U. Stern and D. L. Dill. Parallelizing the murφ verifier. In *Proceedings of Computer Aided Verifier CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 1997.

[63] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[64] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2001.

[65] Kenneth J. Turner. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

[66] A. Valmari. A stubborn attack on state explosion. *Formal Methods in Systems Design*, 1(4) :297–322, 1992.

[67] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[68] F. Wagner. *Modeling Software with Finite State Machines : A Practical Approach.* Auerbach Publications, 2006.

[69] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths (extended abstract). In *FOCS*, pages 185–194, 1983.

[70] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *AOSD '09 : Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172, New York, 2009. ACM.