République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri – Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

Année : **2011**
N°d'ordre :
Série :

Thèse

En vue de l'obtention du diplôme de Doctorat en Sciences en Informatique

Présenté par

BOUNEB ZINE EL ABIDINE

Thème

Vérification symbolique des systèmes critiques : Approche Distribuée

Soutenu le : **03/03/2011** devant le jury composé de :

Dr Chaoui Alloua	Président	M.C.A	U. M. Constantine
Pr Djamel-Eddine SAIDOUNI	Rapporteur	Prof	U. M. Constantine
Dr Boudour Rachid	Examinateur	M.C.A	U. BM Annaba
Dr Bourahla Mustapha	Examinateur	M.C.A	U. M'sila
Dr Merniz Salah	Examinateur	M.C.A	U. M. Constantine

Thèse préparée au Laboratoire MISC, Université Mentouri, 25000 Constantine, Algérie

Remerciements

Je remercie en premier lieu les membres du jury : Dr Chaoui Alloua, qui a accepté la lourde tâche d'être président du jury, Pr Djamel-Eddine Saïdouni , qui a accepté d'être rapporteur de ma thèse, Dr. Boudour Rachid, Dr. Bourahla Mustapha et Dr. Merniz salah , pour le temps qu'ils ont accordé à l'examen de ce travail en dépit de leurs nombreuses activités.

Je remercie tout particulièrement Pr. Djamel-Eddine Saïdouni qui m'a encadré durant cette thèse. Il a été toujours disponible pour répondre aux questions que je lui posais, qu'elles soient théoriques ou techniques. Son enthousiasme, sa patience et sa disponibilité ont beaucoup facilité et agrémenté mon travail. Sans oublier de le remercier de m'avoir accueilli dans son équipe de recherche CFCS du laboratoire MISC.

Résumé

Le modèle des systèmes de transitions étiquetées maximales (STEMs) est un modèle de vrai parallélisme. Ce modèle a été utilisé comme modèle sémantique des algèbres de processus et des réseaux de Petri. Les sémantiques de ces modèles sont définies de manière opérationnelle, ce qui signifie que la génération de STEMs associés aux spécifications se fait de manière automatique. En plus de la prise en compte de la non atomicité temporelle et structurelle des actions dans ce modèle (les actions peuvent avoir des durées non nulles et peuvent être raffinées), un autre intérêt du modèle réside dans son utilisation pour la vérification de propriétés aisément exprimables en CTL et qui portent sur le comportement parallèle des actions. Il est à noter, comme pour la génération des systèmes de transitions étiquetées, la génération des STEMs n'échappe pas au problème classique de l'explosion combinatoire du graphe d'état. Afin d'atténuer les conséquences de ce problème sur la taille des STEMs, différentes solutions ont été proposées et qui visent à réduire la taille des STEMs soit par des représentations réduites modulo certaines relations d'équivalence ; soit par une représentation compacte en termes de BDDs. Quoi que les réductions observées soient importantes, les approches proposées restent centralisées. Par conséquent, le problème de l'insuffisance de la mémoire et celui du temps de calcul des STEMs générés restent un véritable obstacle pour les systèmes de tailles importantes.

Dans cette thèse nous proposons une combinaison de ces approches dans le but de faire passer à l'échelle leurs capacité de vérification, donc nous proposons un algorithme distribué pour la génération de STEMs réduits modulo l'alpha équivalence associés à des spécifications écrites en Basic LOTOS le résultat étant des STEMs réduits équivalents distribués sur tous les sites du réseau. A travers une mise en œuvre de l'algorithme sur un réseau local, nous montrons que la charge des nœuds du réseau se réduit considérablement. Par ailleurs, pour compléter cette approche, nous proposons un algorithme distribué de vérification sur les fragments du STEM réparti sur les nœuds du réseau.

Mots-clés Spécification formelle, Vérification formelle, Langage LOTOS, Sémantique de maximalité.

Table des matières

Table des figures

Première partie

Problématique

Chapitre 1

Introduction

Dans ce chapitre, nous introduisons le contexte de la thèse et nous discutons les résultats trouvés. Le chapitre commence par une brève description de la motivation et le contexte de notre travaille. Puis il passe à la description de la méthode de vérification basée modèle et discute les tendances de la recherche actuelle. Après cette introduction générale, nous donnons un aperçu de la thèse ainsi que les contributions qui seront développées le long de cette dernière.

1.1 Motivation

Les applications réparties, tels que les protocoles de communication et les systèmes distribués, sont caractérisées par une grande complexité. Le développement de ces applications nécessite la considération des contraintes de coopération interprocessus qui exigent la prise en compte de l'indéterminisme, la synchronisation ainsi que certaines propriétés qualitatives tel que l'absence de l'interblocage et de la famine. De même, à cause de leur caractère critique, ces applications sont souvent soumises à de sévères exigences de fiabilité, visant la qualité "zéro erreur".

Les besoins des utilisateurs ainsi que l'environnement accueillant le futur système sont par nature semi formalisables car ils font partie du monde réel, et qui sont liés à des habitudes ou des opinions parfois mal conceptualisés et souvent subjectives (mais qui doivent bien sûr être prises en compte). Même si nous supposons que les besoins des utilisateurs ont été analysés et compris de manière satisfaisante, l'activité qui consiste à établir une spécification du futur système garde un caractère empirique, il s'agit de construire une description de certains aspects du monde réel et il est bien connu qu'une telle entreprise implique forcément, simplification et schématisation. Ainsi, il faut toujours avoir à l'esprit que ce n'est pas parce qu'une spécification est formelle qu'elle ne contient pas de fautes. Ces fautes peuvent provenir d'une mauvaise compréhension des besoins de l'utilisateur ou d'une erreur dans l'expression formelle de ces besoins, d'où la nécessité de l'introduction des techniques de vérification formelle.

1.2 L'ordinateur et les Bugs

Les méthodes formelles sont maintenant utilisées pour les logiciels intervenant dans des systèmes critiques, dont certaines défaillances peuvent être catastrophiques. Citons à titre indicatif quelques uns des exemples les plus célèbres.

Fusée Ariane 5 : Ariane 5 a explosé lors de son premier lancement. En effet, elle a hérité des parties de code de son prédécesseur, Ariane 4, sans qu'une vérification adéquate soit faite [Gle96].

Therac-25 : Therac-25 était une machine de thérapie qui utilise des radiations pour faire les traitements des malades. Dû à une erreur logicielle, six personnes ont trouvé la mort à cause de surdoses [LT93].

Pentium FDIV : Une erreur de conception dans une unité de division à virgule flottante (plus spécifiquement, l'instruction FDIV) d'un processeur Pentium mené à des résultats erronés. Intel été forcé de remplacer tous les processeurs défectueux [FDI94].

Malgré que certains bugs causent des désagréments financiers (par exemple, vol d'espace, conception de processeur, l'échange de stock, contrôle de téléphone), d'autre touchent la sécurité humaine, c'est à dire, menacent la santé ou même la vie humaine (systèmes médicaux, contrôleurs embarqués dans les véhicules, contrôleurs des réacteurs nucléaires). Il est donc crucial de vérifier correctement les fonctionnalités de ces systèmes. Malheureusement, les systèmes qui sont critiques en termes de sécurité, ont habituellement des caractéristiques typiques de systèmes qui sont très difficiles à concevoir correctement, exemple :

- **Un Système embarqué** : Un système n'est pas un ordinateur "autonome", mais le processeur est plutôt embarqué dans un système physique plus grand et contrôle des parties mécaniques de ce dernier.
- **Un Système Réactif :** Un système réactif n'a pas de comportement d'entrée-sortie d'un ordinateur "classique" programmé, mais plutôt s'exécute indéfiniment et réagit aux événements de l'environnement externe.
- **Un Système Concurrent :** Un système concurrent n'est pas un programme séquentiel seul, mais plutôt un ensemble de threads concurrents et qui évoluent selon des schémas de synchronisations complexes.
- **Un Système temps réel :** Un système temps réel interagit avec son environnement selon des contraintes de temps prédéfinies, la temporisation exacte est souvent critique pour le bon fonctionnement du système.

Toutes ces caractéristiques font que ces systèmes sont plus difficiles à concevoir et à vérifier en comparaisons avec les systèmes classiques. Prendre en considération la nature de la sécurité critique de ces systèmes mène à la difficulté de leur vérification, d'où la nécessité de méthodes de vérification fiables et puissantes. Ces méthodes reposent sur l'utilisation de modèles formels de spécification ayant une syntaxe bien définie accompagnée d'une sémantique rigoureuse

qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte.

1.3 Vérification formelle

Il existe essentiellement deux approches dans le domaine de la vérification formelle : Celle basée sur la preuve de théorèmes (theorem-proving) et celle basée sur les modèles (model checking). Ces deux approches ont été étudiées intensivement dans la littérature et de nombreux algorithmes et outils ont été développés. Les méthodes basées sur la preuve de théorèmes permettent de traiter des systèmes ayant un nombre infini d'états, mais elles ne peuvent pas être complètement automatisées, ainsi, ces preuves sont longues et leur établissement doit nécessairement être assisté par un démonstrateur de théorèmes et exige la présence d'un expert humain. En revanche, les techniques basées sur les modèles, bien que restreintes à des systèmes ayant un nombre fini d'état, permettent une vérification simple et efficace. Cette vérification précoce est particulièrement utile dans les premières phases du processus de conception où les erreurs sont susceptibles d'être plus fréquentes.

Dans l'approche basée sur les modèles, l'application à vérifier est d'abord décrite dans un langage de spécification formel de haut niveau dont la sémantique décrit tout le comportement du système à spécifier. Dans le cadre de notre étude, nous avons choisi la technique de description formelle LOTOS. Cette description sera traduite vers un modèle sous jacent représenté par un graphe (ou automate) appelé système de transitions étiquetées maximales (STEM) contenant, éventuellement avec certaines abstractions, tous les comportements possibles du système. Le bon fonctionnement de l'application, étant vérifié à travers l'expression des propriétés attendues en logique temporelle et la vérification de la satisfiabilité de ces propriétés sur le STEM associé à la spécification à l'aide d'outils appelés évaluateurs (model checking).

1.4 Problème de l'explosion combinatoire du graphe d'états

Le facteur principal de limitation du model checking est le problème de l'explosion combinatoire de l'espace d'états. L'utilisation du modèle des systèmes de transitions étiquetées maximales n'échappe pas à ce problème. Plusieurs solutions partielles à ce problème existent ; tel que la combinaison des techniques de vérification avec des techniques de manipulation de graphes. Parmi les plus représentatives, nous pouvons mentionner :

- L'approche qui se base sur la génération d'un modèle réduit modulo certaines relations d'équivalences. Le choix d'une relation d'équivalence est conditionné par les propriétés à préserver après réduction. Parmi ces relations, nous pouvons citer les équivalences de traces, de test, de bissimulation et d'ordre partiel.

- La deuxième approche consiste à coder le modèle par des représentations compactes tel que les diagrammes de décisions binaires (BDDs). Les travaux antérieurs sur le modèle des systèmes de transitions étiquetées maximales ont concernaient les deux approches sus citées. En effet, deux relations d'équivalences ont été définies et utilisées pour la génération à la volée de systèmes de transitions étiquetées maximales réduits, il s'agit de la relation des pas couvrant maximaux et de l'alpha-équivalence. D'autre part, une codification de systèmes de transitions étiquetées maximales en termes de BDDs a été proposée. Bien que ces approches réduisent de manière significative les tailles des systèmes de transitions étiquetées maximales, leur implémentation sur un système mono post pose toujours le problème de l'insuffisance de la mémoire du système utilisé ainsi que le temps considérable pour la génération de ces systèmes de transitions étiquetées maximales. Ce qui limite l'utilisation de ces approches à des applications relativement simples [BCM+92a].

- Afin de surmonter les limitations matérielles, une troisième approche est largement étudiée actuellement. Cette approche consiste en l'utilisation d'un cluster ou d'un réseau de postes de travail. L'intérêt de cette troisième approche réside dans l'adaptation des algorithmes implémentant les deux approches précédentes aux systèmes distribués. Ce qui préserve les résultats des approches précédentes tout en augmentant leurs performances.

Cependant, l'expérience pratique a montré qu'aucune de ces techniques prises séparément ne peut suffire pour éviter l'explosion combinatoire du graphe d'états, et leur utilisation conjointe a de grandes chances d'être plus efficace.

La vérification distribuée. Les dernières années ont été riches en résultats concernant le calcul scientifique de haute performance, qui n'a cessé d'accroitre son importance pour le support d'explorations scientifiques et d'ingénierie multi disciplinaire. Etant donné que les communautés du calcul scientifique et des méthodes formelles n'ont traditionnellement pas travaillé ensemble, il existe très peux de travaux décrivant les interactions possibles entre les différents résultats théoriques et pratiques. Un exemple de telles interactions est l'utilisation des méthodes formelles pour vérifier des systèmes logiciels et matériels pour le calcul de haute performance.

Notre étude concerne le problème inverse, à savoir, l'exploitation de la puissance de calcul et du parallélisme potentiel d'une pile de processeurs pour l'exécution d'applications de vérification.

L'importance scientifique de la vérification distribuée repose sur le passage à l'échelle (ou extensibilité) de la vérification basée sur les modèles comme un outil d'analyse de systèmes logiciels de très grandes tailles. En termes d'états explicites, la barrière du 1 Giga (10^9) états a été dépassée récemment et l'objectif avec les nouvelles méthodes distribuées est de pouvoir manipuler des espaces contenant 1 Téra (10^{12}) états [BCM+92a]. En rendant le processus de vérification plus performant, faisable et applicable dans un contexte réel, un objectif

de ce travail de thèse est également de rendre l'application de la vérification basée sur les modèles plus populaire et largement diffusée. Dans ce but, nous cherchons à simplifier autant que possible le développement des outils de vérification, qui sont des composants logiciels complexes, en promouvant des technologies génériques, applicables à plusieurs problèmes de vérification, et des architectures modulaires de logiciel permettant une réutilisation maximale des algorithmes existants.

Contexte de la thèse

Notre travail de thèse vise à concevoir et construire une infrastructure pour la vérification distribuée basée sur la sémantique de maximalité permettant la vérification de systèmes concurrents complexes. Ce travail s'inscrit dans le contexte de l'environnement FOCOVE (FOrmal COncurrency Verification Environment) pour la vérification des protocoles de communication et des systèmes distribués. Dans la mesure du possible, nous essayons de valider nos propositions par le développement d'outils. Notre travail est constitué principalement des éléments suivants :

- **Un modèle d'architecture à mémoire distribuée** : Les performances des algorithmes distribués dépendent étroitement de l'architecture sur laquelle ils sont exécutés. Les architectures distribuées visées ici ne disposent pas d'une mémoire partagée et sont très génériques, car elles sont 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 une grappe de stations de travail (par exemple une grappe de PCs), dont la taille peut varier d'une échelle locale à une échelle nationale voir internationale, avec des interconnexions de grappes, appelées grilles. Cette architecture implique généralement un mécanisme de distribution des tâches et des données par passage de messages qui nécessite un système de gestion efficace des communications engendrées par le calcul entre les nœuds.

- **Un modèle de programmation de type programme unique et données multiples (SPMD) :** Chaque nœud exécute une instance de l'algorithme distribué qui décrit l'ensemble des tâches à exécuter. Les données sont fournies par la construction locale et progressive de l'espace d'états.

- **Une communication non bloquante asynchrone :** Afin de permettre un recouvrement maximal de la communication par les calculs, les opérations de communication sont rendues non-bloquantes et asynchrones, c'est à dire ne bloquant par l'opération d'émission ou de réception jusqu'à sa terminaison, et ne dépendent pas de la synchronisation avec une opération de communication distante tel qu'une réception lors d'une tentative d'émission. La communication est ainsi rendue concurrente vis à vis des autres opérations du calcul distribué global et local. Le point critique de la vérification basée sur les modèles étant la consommation mémoire. La couche de communication sur

laquelle reposent les algorithmes distribués est basée sur la notion de tampons de communication bornés, qui sont gérés explicitement au niveau algorithmique, et permettent ainsi un contrôle fin de l'utilisation de la mémoire.

- **Une complexité linéaire à taille du problème :** Afin d'assurer la faisabilité de la génération des STEMs et le processus de vérification sur ces dernier, les algorithmes distribués ont une complexité linéaire en temps de calcul et en mémoire par rapport à la taille du problème à résoudre. De plus, et afin de limiter la proportion de messages échangés lors du calcul distribué sur le réseau et limiter ainsi le surcoût en temps induit par le passage de messages, la complexité en nombre de messages transmis est également linéaire par rapport à la taille du problème à résoudre. Une génération de diagnostic (exemple ou contre exemple) distribué. Lors du processus de vérification d'un système logiciel, la non satisfiabilité d'une propriété sur ce système conduit au besoin de décrire un contre exemple illustrant cette situation afin de comprendre la source de l'erreur et de corriger le logiciel en conséquence. Ce mécanisme reste valable dans le cas symétrique où il est désirable d'avoir un exemple justifiant que la propriété est vérifiée. Les algorithmes distribués proposés permettent de fournir un tel diagnostic sous forme d'une représentation implicite distribuée (c'est à dire une fonction successeur définie localement au niveau du nœud responsable de la variable courante du diagnostic).

- **Une détection distribuée de la terminaison** : Les algorithmes distribués proposés contiennent des mécanismes de détection de la terminaison partielle de la génération pour un fragment donné appartenant à un nœud quelconque. Cette approche permet d'obtenir un degré de parallélisme important de la génération en autorisant les parcours distribués de portions minimales et suffisantes d'un fragment parallèlement à la génération d'autres fragments. La détection sur l'ensemble des nœuds de l'inactivité de la génération locale d'un fragment permet ainsi d'accélérer la génération globale du STEM en augmentant le nombre de fragments explorés, et par voie de conséquence la possibilité de trouver plus rapidement un fragment terminal, ne dépendant pas d'autres fragments, à partir des quels les valeurs stables seront propagées.

Plan du document

La thèse est organisée en trois parties principales.

- La première partie, constituée d'un seul chapitre, introduit la problématique sujette de cette thèse. Dans ce chapitre, le problème de l'explosion combinatoire de l'espace d'états sera mis en évidence.

- La deuxième partie comporte un état de l'art sur la vérification distribuée et elle regroupe trois chapitres

- Le Chapitre 2 introduit la sémantique de maximalité du langage LOTOS, et évoque quelques relations d'équivalence définies sur les systèmes de transitions étiquetées maximales, à savoir la $\alpha-$équivalence et la bissimulation forte.
- Le Chapitre 3 introduit la notion de model checking. La logique adoptée étant la logique temporelle CTL, et le modèle est celui des systèmes de transitions étiquetées maximales. L'accent est mis sur l'apport de la sémantique de maximalité pour la vérification de propriétés qu'on ne pouvait pas vérifier dans une approche d'entrelacement. Un algorithme de model checking est présenté.
- Le Chapitre 4 décrit la procédure de la génération et vérification distribuée de l'espace d'états. Dans ce chapitre, nous présenterons le principe de l'algorithme, la manière de distribuer et de représenter le graphe d'états et nous présenterons également les optimisations possibles ainsi que la vérification distribuée des propriétés. Plus particulièrement, nous étudierons la vérification distribuée des propriétés génériques (accessibilité, blocage et vivacité), la vérification distribuée des propriétés décrites en logique temporelle linéaire et la vérification distribuée des propriétés décrites en logique temporelle arborescente.

- La troisième partie comporte nos contributions dans le domaine de la vérification distribuée, elle regroupe trois chapitres :
  - Le Chapitre 5 décrit la construction parallèle des STEMs réduits préservant la $\alpha - \textit{équivalence}$.
  - Le Chapitre 6 présente un algorithme distribué de vérification basée modèle des STEMs en utilisant la logique à trois valeurs, dites de Kleen.
  - Le Chapitre 7 donne une présentation de la mise en œuvre des algorithmes proposés en utilisant le langage Haskell.

Deuxième partie

Notions préliminaires

Chapitre 2

Sémantiques opérationnelles structurées de Basic LOTOS

2.1 Principe de la sémantique de maximalité

La sémantique d'un système concurrent peut être caractérisée par l'ensemble des états du système et des transitions par lesquelles le système passe d'un état à un autre. Dans l'approche basée sur la maximalité, les transitions sont des événements qui ne représentent que le début de l'exécution des actions. En conséquence, l'exécution concurrente de plusieurs actions devient possible, c'est-à-dire que l'on peut distinguer exécutions séquentielles et exécutions parallèles d'actions.

Etant donné que plusieurs actions qui ont le même nom peuvent s'exécuter en parallèle (auto-concurrence), nous associons, pour distinguer les exécutions de chacune des actions, un identificateur à chaque début d'exécution d'action, c'est-à-dire à la transition ou à l'événement associé. Dans un état, un événement est dit maximal s'il correspond au début de l'exécution d'une action qui peut éventuellement être toujours en train de s'exécuter dans cet état là. Associer des noms d'événements maximaux aux états nous conduit à la notion de configuration qui sera formalisée dans la définition 2.2.

Pour illustrer la maximalité et cette notion de configuration, considérons les expressions de comportement E et F de la Figure 2.1. Dans l'état initial, aucune action n'a encore été exécutée, donc l'ensemble des évenements maximaux est vide, d'où les configurations initiales suivantes associées à E et F : ${}_{\phi}[E]$ (état 0) et ${}_{\phi}[F]$ (état $0'$). En appliquant la sémantique de maximalité, les transitions suivantes sont possibles : ${}_{\phi}[E] \xrightarrow{\phi^a x}_m {}_{\{x\}}[stop] \;|||\; {}_{\phi}[b; stop] \xrightarrow{\phi^b y}_m {}_{\{x\}}[stop] \;|||\; {}_{\{y\}}[stop]$

x (resp y) étant le nom de l'événement identifiant le début de l'action a (respectivement b). Etant donné que rien ne peut être conclu à propos de la terminaison des deux actions a et b dans la configuration ${}_{\{x\}}[stop] \;|||\; {}_{\{y\}}[stop]$ (état 3), x et y sont alors maximaux dans cette configuration. Notons que x est également maximal dans l'état (état 1) intermédiaire représenté par la configuration ${}_{\{x\}}[stop] \;|||\; {}_{\phi}[b; stop]$.

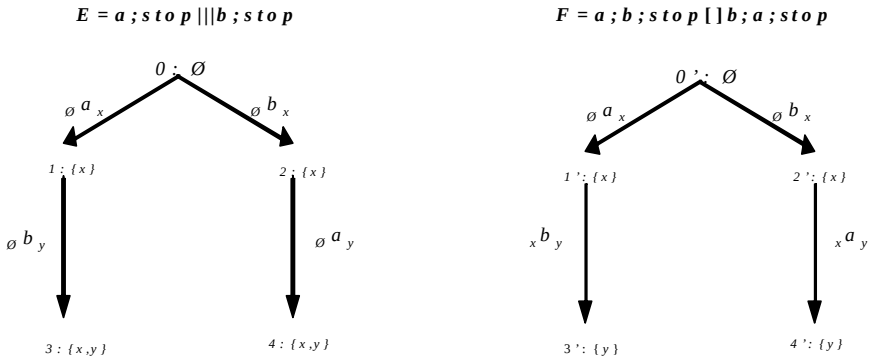


FIG. 2.1 – Arbres de dérivation de E et F

Pour la configuration initiale, associée à l'expression de comportement F, la transition suivante est possible : ${}_{\phi}[F] \xrightarrow{\phi^a x}_m {}_{\{x\}}[b; stop]$. Comme précédemment, x identifie le début de l'action a et il est le nom du seul événement maximal dans la configuration ${}_{\{x\}}[b; stop]$ (état $1'$). Il est clair que, au vu de la sémantique de l'opérateur de préfixage, le début de l'exécution de l'action b n'est possible que si l'action a a terminé son exécution. Par conséquent, x ne reste plus maximal lorsque l'action b commence son exécution ; l'unique événement maximal dans la configuration résultante est donc celui identifié par y qui correspond au début de l'exécution de l'action b. L'ensemble des noms des événements maximaux a donc été modifié par la suppression de x et l'ajout de y, ce qui justifie la dérivation suivante : ${}_{\{x\}}[b; stop] \xrightarrow{\{x\}^b y}_m {}_{\{y\}}[stop]$

La configuration ${}_{\{y\}}[stop]$ est différente de la configuration ${}_{\{x\}}[stop] \;|||\; {}_{\{y\}}[stop]$, car la première ne possède qu'un seul événement maximal (identifié par y), alors que la deuxième en possède deux (identifiés par x et y). Les arbres de dérivation des expressions de comportement E et F obtenus par l'application de la sémantique de maximalité sont représentés dans la figure 2.1

2.1.1 Sémantique de maximalité de Basic LOTOS

Définition 2.1 *L'ensemble des noms des événements est un ensemble dénombrable noté $\mathcal{M}$. Cet ensemble est parcouru par $x, y, ...$ $M, N, ...$ dénotent des sous-ensembles finis de $\mathcal{M}$. L'ensemble des atomes de support Act est $Atm = 2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}$, $2^{\mathcal{M}}_{fn}$ étant l'ensemble des parties finies de $\mathcal{M}$. Pour $M \in 2^{\mathcal{M}}_{fn}$, $x \in \mathcal{M}$ et $a \in Act$, l'atome (M, a, x) sera noté ${}_M a_x$. Le choix d'un nom d'évenement peut se faire de manière déterministe par l'utilisation de toute fonction get : $2^{\mathcal{M}} - \{\phi\} \to \mathcal{M}$ satisfaisant $get(M) \in M$ pour tout $M \in 2^{\mathcal{M}} - \{\phi\}$.*

Définition 2.2 *"Configuration"*

L'ensemble $\mathcal{C}$ des configurations des expressions de comportement de Basic LOTOS est le plus petit ensemble défini par induction comme suit :

- $\forall E \in \mathcal{B}, \forall M \in 2^{\mathcal{M}}_{fn} : \quad {}_M[E] \in \mathcal{C}$

- $\forall P \in PN, \forall M \in 2^{\mathcal{M}}_{fn} : \quad {}_M[P] \in \mathcal{C}$
- *si* $\mathcal{E} \in \mathcal{C}$ *alors hide* L *in* $\mathcal{E} \in \mathcal{C}$
- *si* $\mathcal{E} \in \mathcal{C}$ *et* $F \in \mathcal{B}$ *alors* $\mathcal{E} \gg F \in \mathcal{C}$
- *si* $\mathcal{E}, \mathcal{F} \in \mathcal{C}$ *alors* $\mathcal{E}$ *op* $\mathcal{F} \in \mathcal{C} \quad op \in \{ \; [] \; , \; ||| \; , \; || \; , \; |[L]| \; , \; [> \; \}$
- *si* $\mathcal{E} \in \mathcal{C}$ *et* $\{a_1, ..., a_n\}, \{b_1, ..., b_n\} \in 2^{\mathcal{G}}_{fn}$ *alors* $\mathcal{E}\,[b_1/a_1, ..., b_n/a_n] \in \mathcal{C}$

Etant donné un ensemble $M \in 2^{\mathcal{M}}_{fn}$, ${}_M[\cdots]$ est appelée opération d'*encapsulation*. Cette opération est distributive par rapport aux opérations $[], |[L]|$, *hide*, $[>$ et le renommage des portes. Nous admettons aussi que ${}_M[E \gg F] \equiv {}_M[E] \gg F$. Une configuration est dite canonique si elle ne peut plus être réduite par la distribution de l'opération d'encapsulation sur les autres opérateurs. Par la suite, nous supposons que toutes les configurations sont canoniques.

Proposition 2.1 *[Saï96]"Configuration canonique"*

Toute configuration canonique est sous l'une des formes suivantes ($\mathcal{E}$ *et* $\mathcal{F}$ *étant des configurations canoniques) :*

${}_M[stop]$	${}_M[exit]$	${}_M[a; E]$	${}_M[P]$	$\mathcal{E} \; [] \; \mathcal{F}$
$\mathcal{E} \; \|[L]\| \; \mathcal{F}$	*hide* L *in* $\mathcal{E}$	$\mathcal{E} \gg F$	$\mathcal{E} \; [> \; \mathcal{F}$	$\mathcal{E}\,[b_1/a_1, ..., b_n/a_n]$

Définition 2.3 *La fonction* $\psi : C \to 2^{\mathcal{M}}_{fn}$*, qui détermine l'ensemble des noms des événements dans une configuration, est définie récursivement par :*

$$\begin{array}{lll} \psi\,({}_M[E]) = M & \psi\,(\mathcal{E} \; [] \; \mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) & \psi\,(\mathcal{E} \; |[L]| \; \mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) \\ \psi\,(\mathcal{E} \gg F) = \psi\,(\mathcal{E}) & \psi\,(hide \; L \; in \; \mathcal{E}) = \psi\,(\mathcal{E}) & \psi\,(\mathcal{E} \; [> \; \mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) \\ & & \psi\,(\mathcal{E}\,[b_1/a_1, ..., b_n/a_n]) = \psi\,(\mathcal{E}) \end{array}$$

Définition 2.4 *"Suppression"*

Soit $\mathcal{E}$ *une configuration ;* $\mathcal{E}\backslash N$ *dénote la configuration obtenue par la suppression de l'ensemble des noms des événements* N *de la configuration* $\mathcal{E}$*.* $\mathcal{E}\backslash N$ *est définie récursivement sur la configuration* $\mathcal{E}$ *comme suit :*

$$\begin{array}{ll} ({}_M[E])\backslash N = {}_{M-N}[E] & (\mathcal{E} \; [] \; \mathcal{F})\backslash N = \mathcal{E}\backslash N \; [] \; \mathcal{F}\backslash N \\ (\mathcal{E} \; |[L]| \; \mathcal{F})\backslash N = \mathcal{E}\backslash N \; |[L]| \; \mathcal{F}\backslash N & (hide \; L \; in \; \mathcal{E})\backslash N = hide \; L \; in \; \mathcal{E}\backslash N \\ (\mathcal{E} \gg F)\backslash N = \mathcal{E}\backslash N \gg F & (\mathcal{E} \; [> \; \mathcal{F})\backslash N = \mathcal{E}\backslash N \; [> \; \mathcal{F}\backslash N \\ (\mathcal{E}\,[b_1/a_1, ..., b_n/a_n])\backslash N = \mathcal{E}\backslash N\,[b_1/a_1, ..., b_n/a_n] & \end{array}$$

Définition 2.5 *"Substitution"*

L'ensemble des fonctions de substitution des noms des événements est Subs (i.e. Subs $= \mathcal{M} \to 2^{\mathcal{M}}_{fn}$*) ;* $\sigma, \sigma_1, \sigma_2, ...$ *désignent des éléments de Subs. Etant donnés* $x, y, z \in \mathcal{M}$ *et* $M \in 2^{\mathcal{M}}_{fn}$*, alors*

- *L'application de* σ *à* x *sera écrite* σx *;*

- *La substitution identité ι est définie par $\iota x = \{x\}$;*
- $M\sigma = \cup_{x \in M}\ \sigma x$;
- $\sigma\,[y/z]$ *est définie par :* $\sigma\,[y/z]\,x = \begin{cases} \{y\} & si\ z = x \\ \sigma x & sinon \end{cases}$

Soit σ une fonction de substitution, la substitution simultanée de toutes les occurrences de x dans $\mathcal{E}$ par σx, est définie récursivement sur la configuration $\mathcal{E}$ comme suit :

$$
\begin{array}{ll}
({}_M\,[E])\,\sigma = {}_{M\sigma}\,[E] & (\mathcal{E}\ []\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ []\ \mathcal{F}\sigma \\
(\mathcal{E}\ |[L]|\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ |[L]|\ \mathcal{F}\sigma & (hide\ L\ in\ \mathcal{E})\,\sigma = hide\ L\ in\ \mathcal{E}\sigma \\
(\mathcal{E} \gg F)\,\sigma = \mathcal{E}\sigma \gg F & (\mathcal{E}\ [>\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ [>\ \mathcal{F}\sigma \\
(\mathcal{E}\,[b_1/a_1, ..., b_n/a_n])\,\sigma = \mathcal{E}\sigma\,[b_1/a_1, ..., b_n/a_n] &
\end{array}
$$

Définition 2.6 *"Sémantiques opérationelles structurées de Basic LOTOS"*

La relation de transition de maximalité $\longrightarrow \subseteq \mathcal{C} \times Atm \times \mathcal{C}$ est définie comme étant la plus petite relation satisfaisant les règles suivantes :

1. $\dfrac{}{{}_M[exit] \xrightarrow{M^{\delta}x} {}_{\{x\}}[stop]}\ x = get(\mathcal{M})$

2. $\dfrac{}{{}_M[a;E] \xrightarrow{M^{a}x} {}_{\{x\}}[E]}\ x = get(\mathcal{M})$

3. (a) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}'}{\mathcal{F}\ []\ \mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad \mathcal{E}\ []\ \mathcal{F} \xrightarrow{M^{a}x} \mathcal{E}'}$

4. (a) i. $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad a \notin L \cup \{\delta\}}{\mathcal{E}\ |[L]|\ \mathcal{F} \xrightarrow{M^{a}y} \mathcal{E}'[y/x]\ |[L]|\ \mathcal{F} \backslash M}\ y = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M))$

   ii. $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad a \notin L \cup \{\delta\}}{\mathcal{F}\ |[L]|\ \mathcal{E} \xrightarrow{M^{a}y} \mathcal{F} \backslash M\ |[L]|\ \mathcal{E}'[y/x]}\ y = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M))$

   (b) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad \mathcal{F} \xrightarrow{M^{a}y} \mathcal{F}' \qquad a \in L \cup \{\delta\}}{\mathcal{E}\ |[L]|\ \mathcal{F} \xrightarrow{M \cup N^{a}z} \mathcal{E}'[z/x] \backslash N\ |[L]|\ \mathcal{F}'[z/y] \backslash M}\ z = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - (M \cup N)))$

5. (a) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad a \notin L}{hide\ L\ in\ \mathcal{E} \xrightarrow{M^{a}x} hide\ L\ in\ \mathcal{E}'}$

   (b) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x}{}_m\, \mathcal{E}' \qquad a \in L}{hide\ L\ in\ \mathcal{E} \xrightarrow{M^{i}x} hide\ L\ in\ \mathcal{E}'}$

6. (a) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x}{}_m\, \mathcal{E}' \qquad a \neq \delta}{\mathcal{E} >> F \xrightarrow{M^{a}x} \mathcal{E}' >> F}$

   (b) $\dfrac{\mathcal{E} \xrightarrow{M^{\delta}x} \mathcal{E}'}{\mathcal{E} >> F \xrightarrow{M^{i}x} {}_{\{x\}}[F]}$

7. (a) $\dfrac{\mathcal{E} \xrightarrow{M^{a}x} \mathcal{E}' \qquad a \neq \delta}{\mathcal{E}\ [>\ \mathcal{F} \xrightarrow{M^{a}y} \mathcal{E}'[y/x]\ [>\ \mathcal{F} \backslash M}\ y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

   (b) $\dfrac{\mathcal{E} \xrightarrow{M^{\delta}x} \mathcal{E}'}{\mathcal{E}\ [>\ \mathcal{F} \xrightarrow{M^{\delta}y} \mathcal{E}'[y/x]\ [>\ {}_{\psi(\mathcal{F}) - M}[stop]}\ y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

(c) $\dfrac{\mathcal{F} \overset{M^a x}{\longrightarrow} \mathcal{F}'}{\mathcal{E} \ [> \ \mathcal{F} \overset{M^a y}{\longrightarrow}_{\psi(\mathcal{E})-M}[stop] \ [> \ \mathcal{F}'[y/x]} y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

8. (a) $\dfrac{\mathcal{E} \overset{M^a x}{\longrightarrow} \mathcal{E}' \qquad a \notin \{a_1,...,a_n\}}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \overset{M^a x}{\longrightarrow} \mathcal{E}'[b_1/a_1,...,b_n/a_n]}$

(b) $\dfrac{\mathcal{E} \overset{M^a x}{\longrightarrow} \mathcal{E}' \qquad a = a_i \ (1 \leq i \leq n)}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \overset{M^{bi} x}{\longrightarrow} \mathcal{E}'[b_1/a_1,...,b_n/a_n]}$

9. $\dfrac{P := E \qquad {}_M[E] \overset{M^a x}{\longrightarrow} \mathcal{F}}{{}_M[P] \overset{M^a x}{\longrightarrow} \mathcal{F}}$

L'explication de ces règles-telle qu'elle a été présentée en[SC96]-est donnée ci-dessous, :

La règle 1 implique que la terminaison avec succès ne peut commencer qu'après que toutes les actions référencées par l'ensemble M des noms d'événements aient terminé leur exécution.

La règle 2 caractérise la sémantique des opérateurs de préfixage des actions ; comme le début de l'exécution de l'action a dépend de la terminaison des actions associées aux événements de noms M, seul x, le nom de l'événement associé au début de l'action a, apparaît dans la configuration ${}_{\{x\}}[E]$.

Les règles 3 caractérisent la sémantique de l'opérateur de choix ; c'est une adaptation directe aux configurations de la règle d'inférence définie pour la sémantique opérationnelle structurée de l'entrelacement pour les expressions de comportement de Basic LOTOS.

Les règles 4(a)i, 4(a)ii et 4b caractérisent la sémantique de l'opérateur de composition parallèle. Pour l'explication de ces règles, considérons l'expression de comportement G .

$$G \equiv a; c; d, stop | [c] | b; c; e; stop$$

Dans la configuration ${}_{\varnothing}[G]$, les actions a et b sont sensibilisées et peuvent donc commencer leur exécution ; supposons que $x = get(\mathcal{M})$ et que $y = get(\mathcal{M} - \{x\})$, les dérivations suivantes peuvent être obtenues par l'application de la règle 4.a deux fois de suite :

$${}_{\varnothing}[G] \overset{\varnothing\, a_x}{\longrightarrow}_m \ {}_{\{x\}}[c; d, stop] || [c] |_{\varnothing}[b; c; e; stop] \overset{\varnothing\, b_y}{\longrightarrow}_m \underbrace{{}_{\{x\}}[c; d, stop]}_{S} || [c] | \underbrace{{}_{\{y\}}[c; e; stop]}_{S'}$$

Notons que, dans le cas où b commence son exécution en premier, le nom de l'événement qui lui est associé est x et le nom de l'événement associé à a est y ; la dérivation suivante est possible :

$${}_{\varnothing}[G] \overset{\varnothing\, b_x}{\longrightarrow}_m \ {}_{\varnothing}[a; c; d, stop] || [c] |_{\{x\}}[c; e; stop] \overset{\varnothing\, a_y}{\longrightarrow}_m \ {}_{\{y\}}[c; d, stop] || [c] |_{\{x\}}[c; e; stop]$$

Quand l'action c commence son exécution, elle est identifiée par le nom de l'événement égal à $get(\mathcal{M} - ((\psi(S) \cup \psi(S')) - \{x, y\}) = get(\mathcal{M}) = x$ qui devient le seul nom d'événement présent dans la configuration résultante, ce qui est montré dans la dérivation suivante obtenue par l'application de la règle 4.b :

$$_{\{x\}}[c; d, stop]||[c]|_{\{y\}}[c; e; stop] \overset{\{x,y\}}{\longrightarrow}{}^{bcx}_{m\ \{x\}}\ [d, stop]||[c]|_{\{x\}}[e; stop]$$

Pour cet état, le début de l'action d signifie que l'action c a terminé son exécution, et par conséquent x n'est plus le nom d'un événement maximal ; dans la configuration résultante, le nom x est supprimé des deux arguments de l'opérateur de composition parallèle. L'application de la règle 4.a mène à la dérivation suivante :

$$_{\{x\}}[d, stop]||[c]|_{\{x\}}[e; stop] \overset{\{x\}d_x}{\longrightarrow}{}_{m\quad\{x\}}\ [stop]||[c]|_{\varnothing}[e; stop]$$

Les règles 5a et 5b sont une adaptation directe aux configurations des règles correspondantes de la sémantique de l'entrelacement.

Les règles 6a et 6b traduisent la sémantique de l'opérateur de séquencement de processus. En fait, la règle 6a implique que le comportement de $\mathcal{E} >> \mathcal{F}$ est celui de $\mathcal{E}$ tant que $\mathcal{E}$ n'a pas terminé son exécution avec succès ; alors que la règle 6b stipule qu »une fois que $\mathcal{E}$ a terminé avec succès, l'unique événement maximal dans la configuration résultante, c'est-à-dire $_{\{x\}}[F]$, est alors celui identifié par x.

Les règles 7a, 7b et 7c donnent la sémantique de l'opérateur d'interruption. Afin de mieux comprendre ce mécanisme d'interruption, il est nécessaire de distinguer les différentes stratégies d'interruption qui peuvent être considérées en présence de la non atomicité des actions. Ces stratégies d'interruption sont :

- La première stratégie consiste à interrompre le processus, y compris les actions qui sont en cours. Cette stratégie est illustrée par la Figure 2.2.a.
- La deuxième stratégie consiste à empêcher les actions qui n'ont pas commencé leur exécution de s'exécuter tout en laissant les actions en cours continuer leur exécution. Cette stratégie est illustrée par la Figure 2.2.b.
- La troisième stratégie consiste à n'autoriser l'interruption que lorsque les actions qui sont en cours ne sont terminées. Cette stratégie a pour risque de ne jamais pouvoir interrompre le processus, dans le cas où il y a chevauchement entre le début el la fin des actions. Cette stratégie est illustrée par la Figure 2.2.c.

Dans[Saï96] c'est la deuxième stratégie qui a été prit en considéré.

Dans la sémantique de maximalité, l'ensemble des événements maximaux associé aux états est d'une importance capitale, car il représente les actions qui sont commencé leur exécution et qui peuvent encore être en train de s'exécuter dans cet état. De ce fait, lorsque l'un des processus $\mathcal{E}$ ou $\mathcal{F}$ termine son exécution, de son gré ou suite à une interruption, il est remplacé dans la configuration résultante par le processus qui n'offre plus d'actions nouvelles mais qui garde trace des action qui peuvent encore être en cours. Ceci implique, sous l'hypothèse que les actions ne sont pas atomiques, que l'interruption d'un processus n'affecte que les actions qui n'ont pas commencé leur exécution, les actions en cours terminant leur exécution.

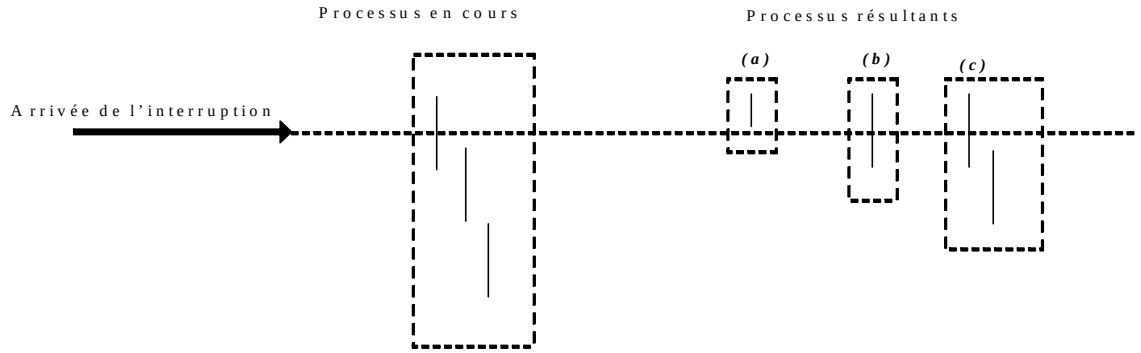


FIG. 2.2 – Stratégies d'interruption

Pour illustrer ces règles, nous prenons l'expression de comportement

$$H \equiv a;(b;exit[>c;exit)$$

En admettant que $x = get(\mathcal{M})$ et que $y = get(\mathcal{M} - \{x\})$, la dérivation suivante est obtenue par l'application de la règle 2

$$_{\varnothing}[H] \xrightarrow{\varnothing\, a_x}_{m\{x\}} [b;exit][>_{\{x\}} [c;exit]$$

De la configuration résultante, nous pouvons déduire que l'action a a commencé son exécution ; pour que les action b et c puissent commencer la leur, il faut que a ait terminé son exécution, ce qui explique les deux dérivations suivantes :

$$_{\{x\}}[b;exit][>_{\{x\}} [c;exit] \quad \begin{cases} \xrightarrow{\{x\}\, b_x}_{m} \ _{\{x\}}[exit][>_{\varnothing} [c;exit] \\ \xrightarrow{\{x\}\, c_x}_{m} \ _{\varnothing}[stop][>_{\{x\}} [exit] \end{cases}$$

En continuant l'application des règles 1,2 et 7 nous obtenons l'arbre de dérivation de la Figure 2.3

Les règles 8 et 9 sont une adaptation directe aux configurations des règles correspondantes de la sémantique de l'entrelacement.

2.1.2 Systèmes de transitions étiquetées maximales

Comme nous l'avons vu dans la section précédente, les arbres de dérivation de la Figure 2.1 et la Figure 2.3 sont des structures obtenus en appliquant directement les règles de la Définition 2.6. Précisément, ces structures représentent le modèle des arbres maximaux [Saï96], c'est un modèle de vrai parallélisme, il nous permet de déterminer pour chacun de ses états les actions qui sont en train de s'exécuter en parallèle ; cela est assuré en associant les informations sur le parallélisme aux états et aux transitions, dont les états contiennent les événements des actions qui sont potentiellement en exécution, et les transitions sont des événements qui ne représentent que le début de l'exécution des actions. En conséquence, nous

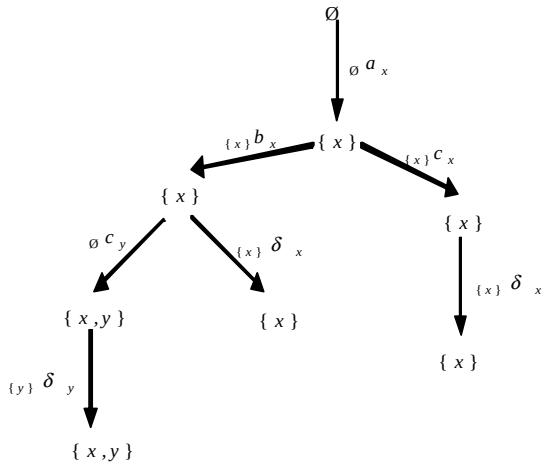


FIG. 2.3 – Arbre de dérivations de H

pouvons facilement différencier le comportement de l'expression " $a; stop|||b; stop$ " de celui de l'expression "$a; b; stop[]b; a; stop$ " (Voir Figure 2.1), ce qu'est impossible pour le cas des modèles d'entrelacement. La différence est entre, par exemple, l'état 3 et l'état 3', dans ce dernier état, il n'y a qu'un seul événement maximal (associé à l'action b). Cependant, l'état 3 contient deux événements maximaux représentant l'exécution parallèle de a et de b.

Définition 2.7 *"Arbre maximal"*

$\mathcal{M}$ *étant un ensemble dénombrable de noms d'événements, un arbre maximal de support* $\mathcal{M}$ *est un quintuplet* (T, l, μ, ξ, ψ) *avec* (T, l) *un arbre étiqueté de support L et*

- $\psi : T \longrightarrow 2^{\mathcal{M}}_{fn}$ *est une fonction qui associe à chaque nœud l'ensemble (fini) des noms des événements maximaux présents au niveau de ce nœud.*
- $\mu : \Theta(T) \longrightarrow 2^{\mathcal{M}}_{fn}$ *est une fonction qui associe à chaque transition l'ensemble (fini) des noms des événements correspondant aux actions qui ont commencé leur exécution et dont la terminaison sensibilise cette transition ; cet ensemble correspond aux causes directes de la transition.*
- $\xi : \Theta(T) \longrightarrow \mathcal{M}$ *est une fonction qui associe à chaque transition le nom de l'événement qui identifie son occurrence.*

tel que, pour tout $(t, tı) \in \Theta(T)$ les conditions suivantes sont satisfaites :

1. $\mu(t, t') \sqsubseteq \psi(t)$: exprime que l'occurrence d'un événement ne dépend qui de l'ensemble d'événements maximaux de nœud précédent

2. $\xi(t,t') \not\subseteq \psi(t) - \mu'(t,t')$: exprime que le nom d'événement attribué à une transition ne peut être le nom d'événement maximal du nœud origine que s'il est une cause directe de cette transition.

3. $\psi(t') = (\psi(t) \backslash \mu(t,t')) \cup \{\xi(t,t')\}$: L'ensemble de nom d'événements maximaux du nœud cible est l'union du nom d'événement associé à la transition et l'ensemble résultant de la restriction de l'ensemble d'événements qui correspondent aux causes directes, sur l'ensemble d'événements maximaux du nœud origine.

une transition de maximalité sur les arbres maximaux peut être représentée par $t \xrightarrow{{}^{N}a_x}_{m} t\prime$avec : $t\prime = te, l(e) = a, \mu(t,t') = \mu({}_{N}a_x) = N, \zeta(t,t') = \zeta({}_{N}a_x) = x$, et $\psi(t') = \psi(t) - N) \cup \{x\}$; les deux conditions suivantes sont satisfaites par la transition (t,t') :$N \subseteq \psi(t)$ et $x \notin \psi(t) - N$.

Les arbres maximaux sont des structures acycliques, en pratique, les graphes d'états sont des graphes cycliques, c'est pour cette raison, les systèmes de transitions étiquetées maximales sont introduits.

Définition 2.8 *"Système de Transitions Etiquetées Maximales "*

$\mathcal{M}$ étant un ensemble dénombrable des noms des évènements, un Système de Transitions Etiquetées Maximales (STEM) de support $\mathcal{M}$ est un quintuplet $(\Omega, A, \mu, \xi, \psi)$ avec :

- *$\Omega :< S, T, \alpha, \beta >$ est un système de transitions tels que :*
  - *S : est un ensemble d'états qui peut être fini ou infini.*
  - *T : est un ensemble de transitions qui peut aussi être fini ou infini.*
  - *α et β sont deux applications de T dans S tels que pour toute transition t de T on a : $\alpha(t)$ l'origine de la transition et $\beta(t)$ son but.*
- *$< \Omega, A >$ est un système de transitions étiquetées par un alphabet A.*

2.1.3 Notions d'équivalence

Dans cette section, deux relations d'équivalence sont évoquées. La première équivalence est une technique de substitution spécifique aux STEMs, connue sous le nom de la $\alpha-$équivalence, et la deuxième relation cherche de regrouper les STEMs qui font les mêmes choix au même moment "bissimulation maximale".

$\alpha-$équivalence

Cette relation permet de regrouper des STEMs, qui décrivent le même comportement dans la seule différence réside dans le choix des noms des événements

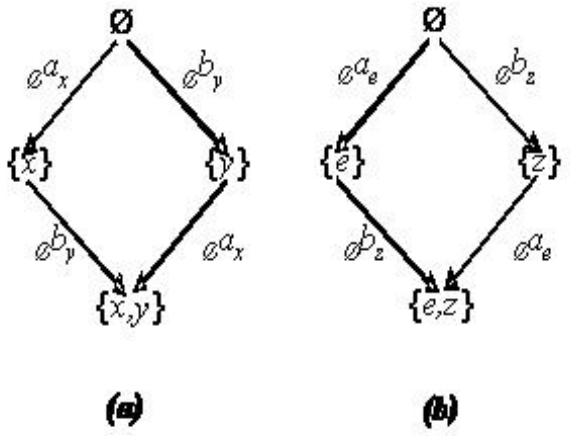


FIG. 2.4 – STEM alpha équivalent

Par exemple, les deux STEM de la Figure 2.4 décrivent le même comportement (l'exécution parallèle de a et de b), nous pouvons obtenir le STEM de la Figure 2.4.(a) à partir de celui de la Figure 2.4.(b) (modulo l'isomorphisme $\cong$)en substituant les noms des événements e (resp z) par x (resp y).

Définition 2.9 *"α-relation" Soit $=_\alpha$ la plus petite relation sur les STEMs telle que $S =_\alpha T$ ssi*

- *$S \cong T$,ou bien*
- *$S \cong \sum_{i \in I} \ {}_{M_i}a_{i\ x_i}\ T_i, T \cong \sum_{j \in J} \ {}_{Mj}a_{j\ xj}\ T_j$, et*

$-\psi(S) = \psi(T)$, et
– il existe une bijection $f : I \longrightarrow J$ telle que, pour tout $i \in I, M = M_{f(i)}, a^i = a^{f(i)}$, et
- $x_i = x_{f(i)}$ and $Ti =_\alpha T_{f(i)}$
- $x_{f(i)} \notin \psi(T_i)$ and $T_i[x_{f(i)}/x_i] =_\alpha T_{f(i)}$

La α-relation est une relation d'équivalence où $=_\alpha \subseteq \cong$

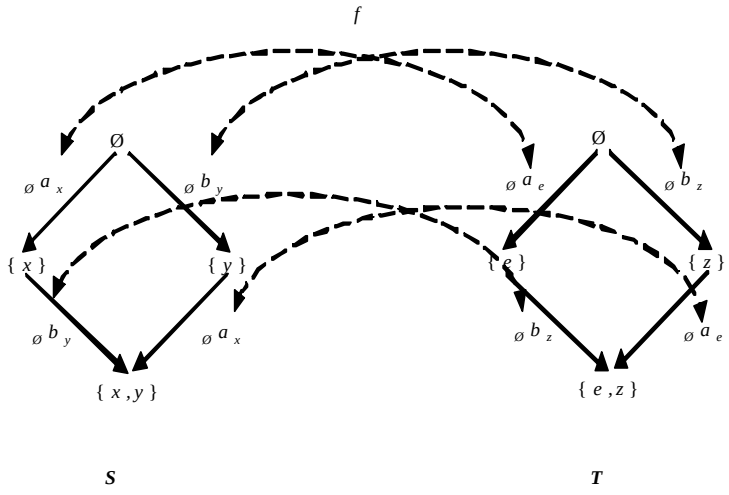


FIG. 2.5 – Une bijection

Propriété 2.1 *Soient T et T' deux STEMs, et soit σ une fonction de substitution ; alors les propriétés suivantes sont vérifiées :*

1. *Si ${}_N a_x T$ est un STEM et $y \notin \psi(T)$, alors ${}_N a_x T =_\alpha {}_N a_x T[y/x]$*
2. $T =_\alpha T\ \iota$
3. $T =_\alpha T' \Longleftrightarrow T\iota \cong T'\iota$
4. $T =_\alpha T' \Longleftrightarrow T\sigma \cong T'\sigma$

La définition 2.9 cherche de trouver une fonction (bijection) entre les deux STEMs afin de décider si ces dernier sont $\alpha-$équivalents ou non. En appliquant cette définition sur l'exemple de la Figure 2.4, la bijection est illustrée dans la Figure 2.5. Une alternative consiste de réduire le test de la $\alpha-$équivalence des STEMs T et T'à un test d'égalité (isomorphisme) des STEMs $T\sigma$ et $T'\sigma$ (Propriété2.1.(4)). prenons le même exemple, il est certain que les STEMs S$[x/x, y/y]$ et T$[x/e, y/z]$ de la Figure 2.5 sont deux STEMs égaux.

Relations de bissimulation maximale

Dans[Saï96], deux relations de bissimulation sont définies, à savoir la relation de bissimulation maximale forte et la relation de bissimulation maximale faible, sont deux relations compatible avec le rafinement d'actions. La première relation est présentée dans cette section.

Définition 2.10 *Soit* $\mathfrak{R} \subseteq STEM \times STEM \times \mathfrak{F}$ *une relation entre STEMs, et soit* $\mathbb{F}_m$ *:*

Rel(STEM) $\rightarrow$*Rel(STEM) une fonction définie comme suit :*$(S, T, F) \in \mathbb{F}^m(\mathfrak{R})$ *ssi :*

1. $Dom(f) \subseteq \psi(S)$ *et* $Cod(f) \subseteq (T)$,
2. *Si* $S \xrightarrow{M} {}^{a_x}_{m} S'$ *, alors il existe* $T \xrightarrow{N} {}^{a_y}_{m} T'$*tel que :*
   - **(a).** *Pour tout* $(u, v) \in f$*, si* $u \notin M$ *alors* $v \notin N$ *; et*
   - **(b).** $(S', T', f') \in \mathfrak{R}$*, avec* $f' = (f\lceil(\psi(S') - \{x\}))\lfloor((T') - \{y\}) \cup \{(x, y)\}$.
3. *Si* $T \xrightarrow{N} {}^{a_y}_{m} T'$ *, alors il existe* $S \xrightarrow{M} {}^{a_x}_{m} S'$*tel que :*
   - **(a).** *Pour tout* $(u, v) \in f$*, si* $v \notin N$ *alors* $u \notin M$ *; et*
   - **(b).** $(S', T', f') \in \mathfrak{R}$*, avec* $f' = (f\lceil(\psi(S') - \{x\}))\lfloor((T') - \{y\}) \cup \{(x, y)\}$.

$\mathcal{R}$ est une relation de bissimulation maximale forte ssi $\mathfrak{R} \in \mathbb{F}^m$ $(\mathfrak{R})$. Si $(S\iota, T\iota, F) \in \mathfrak{R}$ pour une certaine relation de bissimulation maximale forte $\mathfrak{R}$, alors les STEMs S et T sont dits fortement maximalement bissimilaires, ce qui est noté symboliquement par $S \sim_m T$.

- Dans la définition précédente STEM dénote l'ensemble des systèmes de transitions étiquetées maximales, $\mathfrak{F}$ dénote l'ensemble des fonctions ayant pour domaine et codomaine des sous ensembles des évènements maximaux.
- $\mathbb{F}^m$ est une fonction qui a pour domaine et codomaine des ensembles de relations entre STEMs. Elle est utilisée pour la définition par point fixe de la relation de bissimulation maximale.
- La condition (a) stipule que si deux évènements maximaux u et v sont reliés par la fonction f, $(u, v) \in f$; alors si u peut rester maximal dans le futur, alors v doit aussi avoir une possibilité de rester maximal dans le futur. C'est cette condition qui préserve la maximalité .
- La condition (b) montre que f' doit être une extension de f, on enlève x du domaine def', y du codomaine de f' et on ajoute le couple (x, y) à f' .
- L'équivalence de bissimulation maximale est une relation réflexive, symétrique et transitive. L'union de plusieurs relations de bissimulations maximales est une bissimulation maximale. L'union de toutes les bissimulations maximales qui existent entre deux arbres maximaux est le point fixe de $\mathbb{F}^m$.

En appliquant la définition 2.10, nous prouvons que les deus STEMs S et T de la Figure 2.6 sont fortement maximalement bissimulaire, où :
$\mathfrak{R} = \{(1, 1', \varnothing), (2, 2', \{(x, x)\}), (3, 3', \{(x, x), (y, y)\}), (4, 3', \{(y, x)\})\}$.

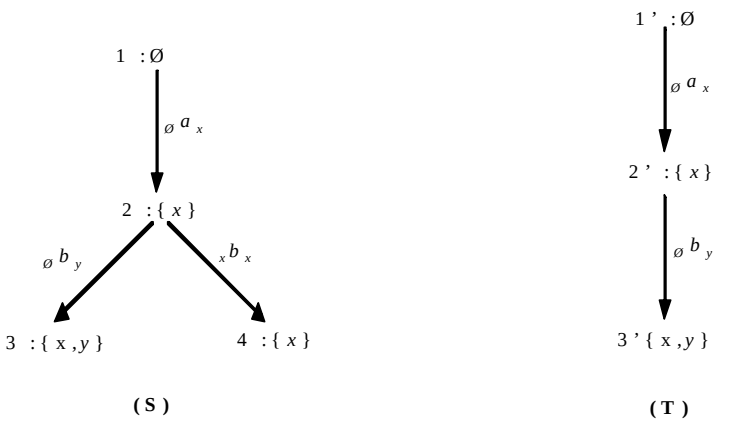


FIG. 2.6 – S et T sont fortement maximalement bissimulaire

2.2 Conclusion

Dans ce chapitre, nous avons présenté l'intuition derrière la sémantique de maximalité et nous avons présenté aussi les systèmes de transitions étiquetées maximales. Le modèle des systèmes de transitions étiquetées maximales peut être utilisé comme une représentation sémantique du comportement du système étudié, d'où la possibilité d'utiliser différents modèles de spécification ; pour cela, il suffit de définir la sémantique en terme de STEMs pour chacun de ces modèles.

Les systèmes de transitions étiquetées maximales sont des modèles de vrai parallélisme. En échappant à l'hypothèse d'atomicité spatiale et temporelle des actions, les transitions de ce modèle sont considérées comme des événements représentant que le début de l'exécution des actions, et l'ensemble d'événements maximaux associé à chaque état correspond aux actions qui sont potentiellement en cours d'exécution. Cela rendre possible d'une part, l'introduction de la sémantique de maximalité aux méthodologies de conception basées sur le raffinement d'actions[SC94] [Saï96], et dans une autre part, la prise en compte du temps[SC03] [BS05].

Chapitre 3

Vérification formelle basée sur la sémantique de maximalité

Dans ce chapitre, nous montrons qu'une adaptation d'une approche de vérification, à savoir l'approche logique, par le passage vers l'utilisation d'une sémantique de vrai parallélisme, est directe.

3.1 Introduction

Par vérification formelle, nous entendons toute technique permettant de confronter un système (sa description opérationnelle) à ses spécifications (aux propriétés que l'on attend de lui). Ce type d'approches nécessite trois ingrédients :

1. Une description opérationnelle du système (son graphe de comportement), générée à partir d'un modèle de spécification.
2. Un langage de spécification permettant d'exprimer les propriétés du système que l'on souhaite vérifier.
3. Une procédure de décision qui permet de contrôler la conformité entre la description opérationnelle du système et sa spécification, c'est-à-dire une procédure qui permet de vérifier que le système satisfait effectivement les propriétés que l'on attend de lui.

La relation entre ces trois éléments est la suivante : Après avoir spécifié formellement un système dans un modèle de spécification, on génère les comportements possibles exprimés dans un modèle sémantique. Ensuite, à l'aide de la procédure de vérification, les propriétés de bon fonctionnement du système peuvent être vérifiées sur le modèle généré. Tout cela est illustré dans la Figure 3.1.

Il existe plusieurs classes de méthodes de vérification formelle divisées selon le moyen utilisé pour exprimer les propriétés attendues du système. On distingue deux grandes classes : celle basée sur l'approche comportementale et celle basée sur l'approche logique.

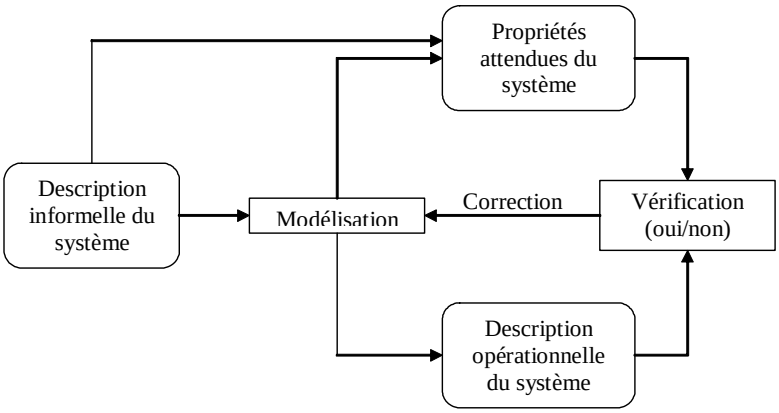


FIG. 3.1 – *Vérification formelle*

L'approche comportementale Dans cette approche, la description opérationnelle du système (son graphe de comportement) ainsi que sa spécification, sont tous les deux exprimés par des comportements. La procédure de décision revient alors à décider de l'équivalence entre deux graphes. De nombreuses relations d'équivalence ont été proposées pour la comparaison et l'analyse des systèmes concurrents, allant de l'équivalence langage à l'équivalence observationnelle, en passant par les modèles de refus et les équivalences de test. Cette diversité s'explique d'une part, par la variété des propriétés spécifiques aux systèmes étudiés, et d'autre part, de la difficulté de définir formellement une sémantique des systèmes de processus.

L'approche logique Dans cette approche, les propriétés attendues du système sont exprimées par des assertions dans une logique, par exemple la *logique temporelle*. Dans ce contexte on distingue deux approches : une basée sur la preuve de théorèmes (*theorem proving* ou *proof checking*) et une autre basée sur le contrôle (évaluation) de modèle (*model checking*).

Méthodes basées sur la preuve

Les méthodes basées sur la preuve consistent à modéliser le programme dans un système formel, puis prouver la correction du programme avec des raisonnements syntaxiques. L'avantage de ces méthodes est qu'elles permettent de traiter des systèmes ayants un nombre infini d'états. En plus, l'intuition humaine peut guider le processus de vérification. En revanche, elles ne peuvent pas être complètement automatisées et nécessitent beaucoup d'effort et d'ingéniosité humaine.

Méthodes basées sur l'évaluation

Les techniques basées sur l'évaluation, bien que restreintes à des systèmes ayant un nombre fini d'états, permettent une vérification (relativement) simple et efficace, qui s'avère particu-

lièrement utile dans les premières phases du processus de conception, quand les erreurs sont susceptibles d'être plus fréquents.

Dans cette classe de méthodes, l'application à vérifier est d'abord décrite dans un langage de spécification de parallélisme de haut niveau ayant une sémantique opérationnelle bien définie. Ensuite, cette description est traduite 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 un certain niveau d'abstraction, tous les comportements possibles du programme. Finalement, les propriétés de bon fonctionnement du système, exprimées dans une logique temporelle sont vérifiées sur le modèle à l'aide de model checkers. Dans ce cas, la procédure de décision consiste à vérifier si la description opérationnelle du système (le graphe de comportement) est un modèle des formules qui expriment les propriétés de correction du système. Cette procédure de décision revient alors à effectuer du «contrôle de modèle» (*model checking*) permettant d'associer à chaque formule l'ensemble des états du modèle qui la satisfont. Ces méthodes offrent l'avantage d'être complètement automatisables, mais souffrent du problème de l'explosion combinatoire d'espace d'états des comportements possibles du système. Pour réduire l'impacte de ce problème, plusieurs solutions ont été proposé : utiliser une sémantique de vrai parallélisme, des structures compactées au niveau de la présentation (*Symbolic Model Checking* [BCM+92b, McM92] utilisant la structure des BDDs [Bry86]) ou des algorithmes de réduction à la volée au niveau génération.

Pour notre part, nous avons adopté une approche logique de vérification utilisant une logique temporelle vu que les logiques temporelles sont bien adaptées pour spécifier les propriétés, car elles permettent d'obtenir des spécifications abstraites et modulaires du système. L'abstraction signifie l'indépendance de la spécification par rapport à toute implémentation : les propriétés requises sont exprimées séparément, sans indiquer la manière dont elles sont implémentées. La modularité signifie qu'une spécification est facilement modifiable en rajoutant, enlevant ou modifiant une des propriétés ; de plus, le processus de vérification sur un modèle peut aussi être effectué de façon modulaire, en vérifiant chaque propriété séparément.

3.2 Model checking

3.2.1 Principe

La démarche du model checking consiste à vérifier si un spécification satisfait une propriété de bon fonctionnement, par la confrontation de l'automate (modèle) associé à cette spécification à une formule de logique temporelle, exprimant la propriété voulue vérifier. On doit noter que la vérification s'effectue sur un modèle du futur système, et non sur le système lui-même.

L'avantage le plus important du model checking est qu'il est complètement automatisable, ce qui a engendré des outils appelés : model checkers. L'architecture globale d'un model checker est illustrée par la Figure 3.2.

Lorsqu'on souhaite vérifier les propriétés d'un système, il est nécessaire de pouvoir les exprimer dans un langage adéquat. La logique temporelle répond à ce besoin. Des formules

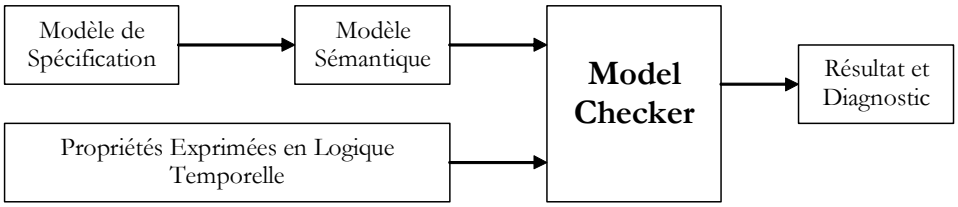


FIG. 3.2 – *Architecture d'un model checker*

simples de la logique temporelle permettent d'exprimer les propriétés usuelles des programmes parallèles.

3.2.2 Logique temporelle

Contrairement à la logique classique, où il existe une fonction d'interprétation unique à partir de laquelle il est possible de déduire la valeur de vérité de chaque formule ; dans la logique temporelle, la fonction d'interprétation est définie sur un graphe. Dans ce graphe, chaque nœud correspond à une fonction classique. A cause de l'introduction d'opérateurs temporels (**A** : quelque soit le chemin, **E** : pour un chemin donné, **G** : toujours, **F** : parfois, **X** : prochain et **U** : jusqu'à), l'interprétation d'une formule dans un nœud ne dépend pas seulement des connaissances de ce nœud, mais peut dépendre des connaissances des autres nœuds du graphe d'interprétation. Les opérateurs précédents sont ceux de la logique temporelle arborescente CTL [CES86, Eme90].

Par exemple dans le graphe de la Figure 3.3, pour que la formule **EG**p soit vraie à l'état s, il faut que p soit vraie au moins dans tous les nœuds d'un chemin sortant de s.

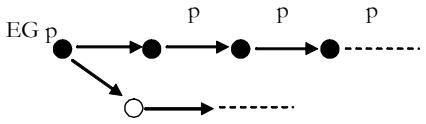


FIG. 3.3 – *La formule* **EG**p *est vraie à l'état initial*

En prenant comme graphe d'interprétation (modèle) le système d'états-transitions associé à un programme, il est possible d'exprimer des propriétés sur les valeurs des formules logiques du programme à chaque état.

3.2.3 Expression de propriétés de bon fonctionnement

A cause de leur aspect critique, les programmes concurrents doivent être de très haut niveau, robustes, et doivent présenter des propriétés de bon fonctionnement. Certaines propriétés assurent que quelque chose de mauvais ne se produira jamais durant l'exécution du système. D'autres propriétés expriment le fait que quelque chose de bon se produira immanquablement

durant l'exécution du système. La première classe de ces propriétés est appelée : *propriétés de sûreté* (propriétés d'invariance). L'autre classe est la classe des *propriétés de vivacité*. Généralement, les propriétés de bon fonctionnement des programmes concurrents sont inclues à l'une des deux grandes classes. Une propriété classique de sûreté est la propriété de l'exclusion mutuelle. Soit un ensemble de processus utilisant un objet commun. Cet objet est appelé : ressource critique. On dit que ces processus sont en exclusion mutuelle lorsque chacun d'eux a une section critique dans son programme. A un instant donné, un et un seul processus au plus exécute sa section critique. C'est-à-dire que durant toute l'exécution du système (un ensemble de processus), on ne doit pas avoir le cas où plus d'un processus en train d'exécuter sa section critique. Alors le problème de l'exclusion mutuelle entre deux processus peut être exprimé en logique temporelle comme suit : $\mathbf{G}(\neg(CS_1 \wedge CS_2))$: les deux processus p_1 et p_2 ne peuvent jamais être en même temps en section critique.

Une autre propriété de sûreté est l'absence de blocage dans un ensemble de processus. Cette propriété exprime le fait qu'à un moment donné, au moins un processus n'est pas en attente. Cette propriété peut être exprimée par : $\mathbf{G}(prêt_1 ... prêt_n)$.

Les propriétés de vivacité signifient qu'après l'initialisation du système, alors dans cet état et ans tous les états suivants, si un requête est lancée, cette requête sera satisfaite plus tard. Ces propriétés sont exprimées en général sous la forme : $\mathbf{G}(req_i \Rightarrow \mathbf{F}\ done_i)$. La terminaison est un exemple d'une propriété de vivacité. Elle exprime que toute opération qui commence son exécution doit se terminer à un moment donné. On peut exprimer cela en logique temporelle comme suit : Pour un processus P_i : $\mathbf{G}(atC_i \Rightarrow \mathbf{F}\ end_i)$: si un processus P_i exécute un opération C_i, alors il doit la terminer à un instant donné.

Une autre classe de propriétés de vivacité est la classe des *propriétés d'équité*. Quand un processus demande l'entrée dans sa section critique, son désir sera satisfait éventuellement à un moment donné, on est ici devant une propriété d'équité forte (réponse à la persistance), on peut exprimer cela par : $\mathbf{GF}(req_i) \Rightarrow \mathbf{GF}(done_i)$. Maintenant, si un processus est prêt à être exécuté, alors il sera exécuté à un moment donné, c'est un exemple de propriété d'équité faible (réponse à l'insistance) : $\mathbf{FG}(req_i) \Rightarrow \mathbf{GF}(done_i)$.

3.3 Model checking basé sur la sémantique de maximalité

Dans cette section, basée principalement sur les résultats de [SB03a, SB04, SB05], nous nous intéressons à une approche de vérification formelle basée sur une sémantique de vrai parallélisme pour exprimer la concurrence. Dans notre contexte, nous adoptons la sémantique de maximalité. Nous avons vu dans la Section 2.1 que cette sémantique nous permet de distinguer entre exécutions séquentielles et exécutions parallèles d'actions. Ainsi, nous montrons comment adapter un algorithme de model checking pour pouvoir vérifier d'autres propriétés ne pouvant pas être vérifiées dans une approche de vérification basée sur l'entrelacement des actions concurrentes.

3.3.1 Formalisation des STEMs

Une présentation détaillée de la sémantique de maximalité peut être trouvée dans [CS95, Saï96]. Dans cette section, nous nous contentons de rappeler la définition de la structure de STEMs et nous illustrons le concept par un exemple simple.

Définition 3.1 *$\mathcal{M}$ étant un ensemble dénombrable de nom d'événements, un système de transitions étiquetées maximales de support $\mathcal{M}$ est un quintuplet $(\Omega, A, \mu, \xi, \psi)$ avec :*

- *$\Omega = \langle S, T, \alpha, \beta \rangle$ est un système de transitions tels que :*
  - *S : l'ensemble des états dans lesquels peut se trouver le système, cet ensemble peut être fini ou infini.*
  - *T : l'ensemble des transitions indiquant le changement d'états que peut réaliser le système, cet ensemble peut aussi être fini ou infini.*
  - *α et β sont deux applications de T dans S tel que pour toute transition $t \in T$ on a : $\alpha(t)$ désigne l'origine de la transition et $\beta(t)$ son but.*
- *(Ω, A) est un système de transitions étiquetées par un alphabet A.*
- *$\psi : S \longrightarrow 2^{\mathcal{M}}_{fn}$ est une fonction qui associe à chaque état l'ensemble fini des noms des événements maximaux présents au niveaux de cet état.*
- *$\mu : T \longrightarrow 2^{\mathcal{M}}_{fn}$ est une fonction qui associe à chaque transition l'ensemble fini des noms des événements correspondant aux actions qui ont commencé leur exécution et dont la terminaison sensibilise cette transition.*
- *$\xi : T \longrightarrow \mathcal{M}$ est une fonction qui associe à chaque transition le nom de l'événement qui identifie son occurrence.*

Tel que pour toute transition $t \in T$, $\mu(t) \subseteq \psi(\alpha(t))$, $\xi(t) \notin \psi(\alpha(t)) - \mu(t)$ et $\psi(\beta(t)) = (\psi(\alpha(t)) - \mu(t)) \cup \{\xi(t)\}$.

Entre autre, un STEM permet d'exprimer le comportement des systèmes concurrents par la détermination de l'ensemble des actions qui sont potentiellement en cours d'exécution dans chaque état (voir Figure 3.4).

Soit l'expression de comportement Basic LOTOS `H = a;b;stop|||(c;stop[]a;stop)`. Intuitivement, nous pouvons remarquer que durant le comportement d'une telle spécification, avec l'hypothèse que les actions ne sont pas atomiques, dans certains états les actions a et c, a et a, b et c ainsi que b et a peuvent s'exécuter en parallèle. Il est clair que la sémantique d'entrelacement ne permet pas de voir de telles situations. Cependant l'application de la sémantique opérationnelle de maximalité de Basic LOTOS de la Section 2.6 permet la génération du STEM de la Figure 3.4.

On peut constater que les états 2, 3, 4, 5, 7, 8, 10, 11, 13 et 14 représentent de tels cas d'exécutions concurrentes d'actions.

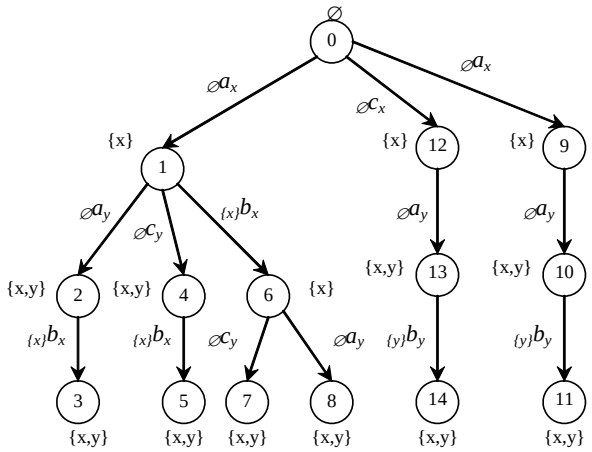


FIG. 3.4 – *STEM de l'expression H*

3.3.2 La logique CTL

CTL est une logique temporelle propositionnelle de branchement utilisée fréquemment dans les techniques de model checking [CE81, BAMP83, CES86, BBL$^+$99]. CTL contient les opérateurs temporels usuels : **X** (le prochain instant), **F** (éventuellement), **G** (toujours) et **U** (jusqu'à) qui doivent être immédiatement précédés par l'un des quantificateurs de chemin qui sont **A** (pour tous les chemins) ou **E** (il existe un chemin). Par exemple, **AG**p est satisfaite dans un état si pour tous les chemins à partir de cet état, p est toujours vrai.

La logique temporelle CTL permet d'exprimer des formules sur les états, notées φ, et des formules sur les chemins, notées ω. Leur syntaxe est comme suit :

$$\varphi ::= p \mid True \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\omega \mid \mathbf{E}\omega$$

$$\omega ::= \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

où $p \in AP$ est une proposition atomique.

Nous pouvons distinguer huit opérateurs de base dans la logique CTL : **AX**, **EX**, **AG**, **EG**, **AF**, **EF**, **AU** et **EU** définis comme suit :

- **AX**f est satisfaite dans un état si la formule f est satisfaite dans tous ses successeurs.
- **EX**f est satisfaite dans un état si au moins un de ses successeurs satisfait la formule f.
- Un état satisfait **AF**f si sur chaque chemin issu de cet état, il y a au moins un état qui satisfait f.
- Un état satisfait **EF**f s'il existe un chemin issu de cet état contenant au moins un état qui satisfait f.

- Un état satisfait $\mathbf{AG}f$ si sur tous les chemins à partir de cet état, f est toujours satisfaite.
- Un état satisfait $\mathbf{EG}f$ s'il existe un chemin issu de cet état où f est toujours satisfaite.
- $\mathbf{A}(f\mathbf{U}f')$ est satisfaite dans un état si sur tous les chemins à partir de cet état f est toujours vérifiée jusqu'à un état qui vérifie f'.
- $\mathbf{E}(f\mathbf{U}f')$ est satisfaite dans un état s'il existe un chemin à partir de cet état où f est toujours vérifiée jusqu'à un état qui vérifie f'.

3.3.3 Sémantique de maximalité et vérification logique

Type de propriétés à vérifier

Dans la section précédente nous avons vu le genre de propriétés qu'on pouvait exprimer en utilisant la logique temporelle CTL ; le raisonnement portait sur des propositions logiques appartenant aux états du système, et les propriétés qu'on voulait vérifier sont généralement divisées en deux grandes classes qui sont : la classe des propriétés de vivacité et celle des propriétés de sûreté [Lam83]. Evidemment, ce sont des propriétés classiques bien connues et plusieurs model checkers ont été développés pour les vérifier.

Pour notre part, grâce à la sémantique de maximalité et l'utilisation du modèle des STEMs, les informations incluses dans les états du modèle représentent les actions qui sont potentiellement en cours d'exécution. De ce fait, on peut exprimer des propriétés appartenant au classes citées antérieurement telles que *l'exclusion mutuelle* de manière plus naturelle, ainsi que de nouvelles propriétés qui portent sur les actions et leur exécution parallèle. L'expression de ces propriétés ne nécessite pas l'utilisation d'une nouvelle logique ou l'introduction de nouveaux opérateurs, car on peut utiliser la logique temporelle arborescente CTL et considérer les actions dans les états comme étant des formules atomiques. Cependant, ce qui change c'est l'intuition derrière les formules. A titre d'exemple, la formule $\mathbf{EF}\,(a \wedge b)$ où a et b sont des noms d'actions, signifie qu'il existe au moins un chemin dans lequel l'exécution en parallèle de a et b peut avoir lieu, de manière similaire on peut expliquer intuitivement toutes les formules de la logique CTL dont le modèle sur lequel elles vont être vérifiées est un STEM comme suit :

- $a \wedge b$ dans un état S signifie que a et b peuvent être exécutées en parallèle dans l'état S.
- $\neg a$ dans un état S signifie que l'exécution de a dans l'état S ne peut pas avoir lieu.
- $\mathbf{EX}a$ dans un état S_0 signifie qu'il existe au moins un chemin $(S_0, S_1, ...)$ où a pourra s'exécuter dans l'état S_1.
- $\mathbf{AX}a$ dans un état S_0 signifie que quelque soit le chemin $(S_0, S_1, ...)$ issu de l'état S_0, a pourra s'exécuter à l'état S_1.

- $\mathbf{E}[a\mathbf{U}b]$ dans un état S_0 signifie qu'il existe un chemin $(S_0, S_1, ..., S_k, ...)$ où b pourra s'exécuter dans l'état S_k et a pourra s'exécuter dans chaque état de ce chemin qui précède l'état S_k.

- $\mathbf{A}[a\mathbf{U}b]$ dans un état S_0 signifie que quelque soit le chemin issu de l'état S_0, il existe un état appartenant à ce chemin où b pourra s'exécuter et a pourra s'exécuter dans chaque état de ce chemin qui précède cet état.

Ainsi, pour exprimer des propriétés de sûreté ou de vivacité, on n'a plus besoin d'utiliser les formules logiques pour indiquer l'état d'évolution d'un processus, mais on raisonne directement sur les actions. En procédant ainsi, les propriétés seront plus faciles à exprimer et leur signification paraît plus naturelle.

Si nous prenons comme exemple d'exclusion mutuelle le problème des lecteurs-rédacteurs, en supposant qu'il existe une seule variable où un rédacteur écrit une information et un lecteur la lit, on sait que les accès à cette variable sont mutuellement exclusifs, donc au lieu d'exprimer l'exclusion mutuelle comme : $\mathbf{AG}\neg(atC_1 \wedge atC_2)$, on peut l'exprimer tout simplement par $\mathbf{AG}\neg(red\_ecrire \wedge lect\_lire)$, où :

- red_ecrire est l'action d'écriture du rédacteur.

- $lect\_lire$ est l'action de lecture du lecteur.

- atC_1 est une formule logique. Elle est vraie si le rédacteur est dans sa section critique.

- atC_2 est une formule logique. Elle est vraie si le lecteur est dans sa section critique.

Dans la Section 3.3.5, des exemples plus élaborés seront présentés par l'étude du problème du dîner des philosophes.

Remarque 3.1 *On a vu dans la structure des STEMs qu'à chaque action est associé un nom d'événement qui permet de distinguer entre plusieurs actions de même nom qui sont en exécution parallèle. Par conséquent, on peut avoir plusieurs actions de même nom dans un même état mais les noms des événements associés seront différents. En partant de ce point, on peut envisager des formules qui nous permettront de raisonner sur le nombre d'occurrences d'une action qu'on peut avoir en parallèle, c'est-à-dire vérifier le degré de l'autoconcurrence d'une action.*

On peut utiliser la notation $a : 5$ pour dire qu'on peut avoir 5 occurrences de l'action a en parallèles, $a : 5$ sera donc considérée comme étant une proposition atomique ; ce qui évitera l'introduction d'un nouvel opérateur à la logique. On aura alors deux formes de propositions atomiques, soit de la forme a soit de la forme $a : n$ où n est une valeur entière.

En s'appuyant sur ces aspects intuitifs et en utilisant cette dernière notation, on peut exprimer de nouvelles propriétés telles que :

L'incompatibilité de deux actions On peut exprimer que a et b sont incompatibles par la formule

$$\mathbf{AG}\neg(a \wedge b)$$

c'est-à-dire qu'elles ne pourront jamais s'exécuter en parallèle. De manière similaire, vérifier que des actions peuvent s'exécuter en parallèle s'exprime comme suit :

$$\mathbf{EF}\,(a \wedge b \wedge ... \wedge z)$$

où $a, b, ...$et z sont des noms d'actions.

Le degré d'autoconcurrence d'une action Par exemple $\mathbf{EF}\,(a : n)$ est vraie s'il existe un état dans lequel on a n actions s'exécutant en parallèle et dont le nom est a.

Il est clair que ces propriétés n'auraient pas pu être exprimées sur un modèle d'entrelacement.

3.3.4 Algorithme de model-checking

Après avoir vu le genre de propriétés qu'on pouvait exprimer en utilisant le modèle des STEMs pour la représentation des comportements possibles et la logique temporelle CTL comme langage de spécification des propriétés, dans ce qui suit nous illustrons la méthode d'évaluation des formules de la logique CTL sur le modèle des STEMs à travers l'adaptation de l'algorithme de model checking presenté dans [CES86]. Le choix de cet algorithme est fait à titre d'illustration, il est clair que différents algorithmes de model checking peuvent être adaptés au modèle des STEMs de manière similaire.

Fonctionnement de l'algorithme

Supposons qu'on a une structure (modèle) fini $M = (S, R, L)$, et une formule CTL p_0. Le but est de déterminer les états s de M où on a $M, s \models p$.

Cet algorithme est conçu pour être exécuté en étapes : la première étape traite toutes les sous-formules de p_0 de longueur 1, la seconde étape traite toutes les sous-formules de p_0 de longueur 2, et ainsi de suite... A la fin de la $i^{\text{ème}}$ étape, chaque état sera étiqueté par l'ensemble de toutes les sous-formules de longueur i vraies dans cet état. Pour élaborer l'étiquetage à l'étape i, on a besoin des informations collectées dans les étapes précédentes. Par exemple, l'état s doit être étiqueté par la sous-formule $(q \wedge r)$ exactement si l'état s est étiqueté par q et r.

Pour la sous-formule $\mathbf{A}[q\mathbf{U}r]$, on aura besoin des informations sur les états successeurs de s ainsi que sur l'état s lui-même, puisque $\mathbf{A}[q\mathbf{U}r] = r \vee (q \wedge \mathbf{AXA}[q\mathbf{U}r])$. Initialement, $\mathbf{A}[q\mathbf{U}r]$ est ajoutée à tous les états déjà étiquetés par r. Ensuite, $\mathbf{A}[q\mathbf{U}r]$ va être propagée et ajoutée à tout état étiqueté par q dont tous ses successeurs sont étiquetés par $\mathbf{A}[q\mathbf{U}r]$.

De la même manière on raisonne pour $\mathbf{E}[q\mathbf{U}r]$.

On doit noter que les autres opérateurs modaux sont implicites et définis autant qu'abréviations :

$$\begin{aligned} q \vee r &\equiv \neg(\neg q \wedge \neg r) \\ q \Rightarrow r &\equiv \neg q \vee r \\ q \Leftrightarrow r &\equiv (q \Rightarrow r) \wedge (r \Rightarrow q) \\ \mathbf{AX}q &\equiv \neg\mathbf{EX}\neg q \\ \mathbf{EF}p &\equiv \mathbf{E}(true\mathbf{U}p) \\ \mathbf{AG}p &\equiv \neg\mathbf{EF}\neg p \\ \mathbf{AF}p &\equiv \mathbf{A}(true\mathbf{U}p) \\ \mathbf{EG}p &\equiv \neg\mathbf{AF}\neg p \end{aligned}$$

Algorithme

Entrée Une structure temporelle $M = (S, R, L)$ comme modèle sémantique ; et une formule p_0 écrite en CTL.

Sortie Ensemble d'états de M qui satisfont la formule p_0.

Début
pour $i = 1$ **à** $longueur(p_0)$ **faire**
pour chaque sous-formule p de p_0 de longueur i **faire**
cas forme de p **faire**
$p = P$: une proposition atomique :
/* ne rien faire */
$p = q \wedge r$:
pour chaque $s \in S$ **faire**
si $q \in L(s)$ **et** $r \in L(s)$ **alors**
ajouter $(q \wedge r)$ à $L(s)$;
fsi
fpour
$p = \neg q$:
pour chaque $s \in S$ **faire**
si $q \notin L(s)$ **alors**
ajouter $\neg q$ à $L(s)$;
fsi
fpour
$p = \mathbf{EX}q$:
pour chaque $s \in S$ **faire**
si $\exists$ successeur s' de s / $q \in L(s')$ **alors**
ajouter $\mathbf{EX}q$ à $L(s)$;
fsi

```
        fpour
  p  = A[qUr] :
        pour chaque s ∈ S faire
        si r ∈ L(s) alors
            ajouter A[qUr] à L(s) ;
        fsi
        fpour
        pour j = 1 à Card(S) faire
          pour chaque s ∈ S faire
            si q ∈ L(s) et si ∀ successeur s' de s /
              A[qUr] ∈ L(s') alors
                ajouter A[qUr] à L(s) ;
            fsi
          fpour
        fpour
  p  = E[qUr] :
        pour chaque s ∈ S faire
          si r ∈ L(s) alors
            ajouter E[qUr] à L(s) ;
          fsi
        fpour
        pour j = 1 à Card(S) faire
          pour chaque s ∈ S faire
            si q ∈ L(s) et si ∃ successeur s' de s /
              E[qUr] ∈ L(s') alors
                ajouter E[qUr] à L(s) ;
              fsi
            fpour
          fpour
      fcas
  fpour
  fpour
  Fin
```

Cette version de l'algorithme possède une complexité temporelle linéaire en fonction de la longueur de la formule à vérifier et quadratique en fonction de la taille de la structure M [Eme90].

Puisqu'on a choisi la logique CTL comme langage d'expression des propriétés, et puisque les informations sur lesquelles on veut raisonner sont incluses dans les états du STEM, alors on remarque bien que cet algorithme peut s'adapter à notre étude pour la vérification des

propriétés sur les STEMs. Pour cela, il suffit de considérer les actions dans les états du STEM comme étant des propositions atomiques et rajouter à l'algorithme le cas où p est de la forme $q : n$ comme suit :

cas $P = q : n$:
  pour chaque $s \in S$ **faire**
    si $\exists\ x_1, x_2, ..., x_n\ /\ x_1, x_2, ..., x_n \in L(s)$
      et $act\,(x_1, s) = q$, $act\,(x_2, s) = q$, ..., $act\,(x_n, s) = q$
      et $card\,(\{x_1, x_2, ..., x_n\}) = n$ **alors**
        ajouter $q : n$ à $L(s)$;
    fsi
fin pour

tel que $x_1, x_2, ..., x_n$ sont des noms d'événements, q est une action et act est une fonction qui reçoit un nom d'événement et un état et elle retourne le nom d'action associé à l'événement donné comme paramètre dans cet état.

Cet algorithme adapté est implanté dans l'outil `LotoStem` qui fait partie de l'environnement `FOCOVE` développé dans notre laboratoire.

3.3.5 Exemple

Cette section présente l'étude du problème du dîner des philosophes [Dij71]. Les spécifications formelles détaillées de ce problème en Basic LOTOS peuvent être trouvées dans [SB03b].

Exclusion mutuelle

L'exclusion mutuelle concerne l'utilisation de chaque fourchette (ressource non partageable). Pour deux philosophes, la propriété s'exprime par :

```
A G (not (Philo1Prendf1 and Philo2Prendf1) and
     not (Philo1Prendf2 and Philo2Prendf2))
```

Vous pouvew vérifié le cas de deux, trois et quatre philosophes en utlisant l'outil `LotoStem`, et vous trouverer comme résultat la satisfiabilité des trois spécifications.

Abscence de l'interblocage

Le cas de l'interblocage se produit lorsque chaque processus du système possédant une ressource (fourchette dans cet exemple) demande une ressource déjà allouée à un autre processus. Les requêtes des processus forment ainsi une attente circulaire. L'absence de l'interblocage stipule que le système pourra toujours progresser dans le futur, et ne sera jamais bloqué. Cette propriété s'exprime par : `A G((E X true) or delta)`.

Cette expression CTL signifie que chaque état du système aura au moins un état successeur. Dans le cas contraire, il faut que la dernière action qui est potentiellement en cours

d'exécution dans cet état, selon la sémantique de maximalité, soit l'action *delta* (δ) qui signifie la terminaison d'un processus avec succès. Les résultats obtenus sont les suivants :

- Dans un premier temps nous avons écrit les spécifications répondant à l'hypothèse que les fourchettes sont prises dans le même ordre (élimination d'une des conditions nécéssaires de l'interblocage). Ce résultat est confirmé par l'outil `LotoStem` par la vérification de la satisfiabilité de la formule précédente.
- Dans un deuxième temps, nous avons spécifié la demande des ressources dans n'importe quel ordre. Dans ce cas, la formule n'est pas vérifiée, et des états d'interblocage ont été détectés [SB03b].

Absence de famine

On peut exprimer qu'à chaque fois qu'un philosophe veut manger alors il viendra un moment où il pourra le faire. Pour un philosophe i, l'absence de famine peut être exprimée en CTL par : `A G(Philo_i_VeutManger => A F philo_i_mange)`.

3.4 Conclusion

Dans ce chapitre, et après avoir évoqué succinctement le principe d'une approche de vérification formelle, à savoir la vérification logique, nous avons discuté l'intérêt du modèle des STEMs pour la vérification des propriétés liées au parallélisme. Cet apport a été principalement induit par la présence des informations de maximalité liées aux états et aux transitions. En particulier, nous avons montré comment ces informations facilitent l'écriture de propositions atomiques exprimant les propriétés à vérifier. Une attention particulière a été portée sur la lecture naturelle et intuitive de ces propriétés. A titre d'exemple, nous avons souligné la vérification du degré de parallélisme en général et de l'autoconcurrence en particulier dans une application concurrente donnée. Dans le but de concrétiser cette étude, nous avons montré comment un algorithme classique de model-checking [CES86] peut être adapté de manière directe au modèle des STEMs. Concernant cette étude, nous avons adopté une approche d'évaluation globale.

Pour clarifier les idées, nous avons présenté quelques résultats de la vérification du problème du dîner des philosophes. La validation des résultats a été réalisée par l'utilisation de l'outil `LotoStem` de l'environnement de vérification formelle `FOCOVE.`

Chapitre 4

Approches pour la vérification parallèle et distribuée

4.1 Motivations

Le but principal de l'exploitation d'un environnement parallèle ou distribué pour une vérification basée modèle (model checking) est d'étendre l'applicabilité des algorithmes de ce dernier à des systèmes complexes de tailles importantes. Un super ordinateur parallèle, une grille de calcul ou un réseau d'ordinateurs peuvent fournir les ressources indispensables pour lutter contre le problème de l'explosion combinatoire de l'espace d'états. Le traitement parallèle peut augmente la vitesse de vérification par l'offre de plusieurs CPUs qui travaillent sur le même problème ainsi par l'offre d'une mémoire globale de plus grande taille. Par ailleurs, ces systèmes offrent une bande passante plus large pour l'accès aux données. Il est à noter qu'en présence de problèmes de complexité exponentielle, la performance totale de calcul devient assez intéressante.

Cependant, la vérification distribuée est intrinsèquement un problème difficile. Elle requiert une double expertise à la fois en vérification formelle et en calcul distribué. La vérification, de son côté, peut être divisée en plusieurs étapes, parmi lesquelles, la création d'un modèle abstrait du système à vérifier, la vérification de propriétés sur le modèle et la génération d'un contre exemple dans le cas où les propriétés ne sont pas satisfaites. La distribution de la vérification consiste à traiter essentiellement trois points supplémentaires : Le partage des données et des tâches, la minimisation du temps global d'inactivité, et la détection de la terminaison de l'application distribuée. Les algorithmes distribués résultants de ces étapes sont complexes et difficiles à concevoir.

Très souvent, malgré la disponibilité d'un grand nombre d'ordinateurs parallèles, les développeurs ne peuvent pas tirer profit de cette puissance de calcul. En effet les vérificateurs sont conçus pour être exécutés sur des machines mono processeur.

Une mise en œuvre sainne des outils de vérification sur des architectures parallèles peut

augmenter la vitesse d'accès à la mémoire et d'accélérer le calcul de manière significative. De ce fait, la détection des erreurs de foctionnement des applications à concevoir devient plus rapide.

Le calcul parallèle se base principalement sur la décomposition du calcul global en sous calculs relativement indépendants. Etant donné que les propriétés atomiques peuvent être vérifiées indépendamment, nous pouvant donc les vérifier en parallèle. Par ailleurs, la vérification d'une conjonction de propriétés atomiques s'arrête aussi tôt qu'un nœud détecte la non satisfiabilité de l'une des ces propriétés.

Dans la prochaine section, nous exposons quelques algorithmes de vérification qui ont fait l'objet de distribution ainsi que les tendances actuelles pour la distribution de telles applications.

4.2 Travaux connexes

L'une des caractéristiques principales des algorithmes s'exécutant dans un environnement distribué est la distribution des données sur les nœuds de calcul avec le moindre supplément de calcul dédié à la coordination entre ces nœuds. De ce fait, les algorithmes distribués de model checking n'échappent pas à cette préoccupation, ainsi la distribution des états du modèle sur les nœuds de calcul a un impact directe sur les performances de ces algorithmes. Quoique les algorithmes distribués de model checking partagent la même idée de répartition des états sur les nœuds de calcul, la chose qui les distingue se résume en la manière de partitionner cet espace d'états. Deux facteurs principaux caractérisent le partitionnement du graphe, à savoir le degré de connexité entre les états de chaque partie du graphe (ensemble des états stockés sur un nœud) et l'équilibrage de la charge entre les nœuds (stockage et calcul).

Le degré de connexité est mesuré par les relations de transition entre les états d'une même partie. Cependant, on parle de charge équilibrée lorsque les partie du graphe sont de mêmes tailles et l'effort de calcul est le même pour chaque nœud.

La mise en œuvre des algorithmes de distribution est fonction de plusieures paramètres à savoir :

- Le choix d'une architecture quelle soit à mémoire partagée ou avec passage de messages.
- L'utilisation des tables de hachage ou des arbres binaires pour le stockage des états sur chaque nœud.
- La méthode de partitionnement considérée quelle soit basée sur l'utilisation d'une fonction de hachage statique ou dynamique.

Il est à noter que la plus part des algorithmes de partitionnement proposés dans la littérature se basent sur un partitionnement statique [I.S01],[F.L99],[SD97]. Ce mode de partitionnement est fonction des états et non de la distribution de la charge, ce qui rend difficile l'obtention d'une bonne répartition du graphe sur les nœuds de calcul. Ce constat motive des réflexions sur une répartition dynamique équilibrant la charge entre les nœuds.

Le modèle-checking Murϕ a fait l'objet d'une parallélisation par le découpage des états visités par le biais d'une fonction d'équilibrage de charge randomisée. En d'autre terme l'identité du nœud destinataire d'un état est calculé par une fonction de hachage dont le résultat est une valeur numérique entre deux limites prédéterminés [SD97]. La complexité de tels algorithmes est souvent linéaire.

Dans [F.L99] une implémentation distribuée du model checking SPIN a été proposée. La fonction de hachage randomisée utilisée se base sur la structure des états afin de diminuer les transitions distribuées (transition entre deux états qui sont stockés sur deux nœuds distincts, en anglais "*cross transition*").

Dans [HV00] les résultats de [SD97] concernant la fonction de hachage sont utilisés pour la mise en œuvre distribuée du modèle-checker symbolique UPPAAL : Une heuristique est ajoutée pour ordonner les états sur chaque nœud afin que la recherche distribuée soit aussi proche que possible d'une recherche en largeur (en anglais "*breadth-first search*"). Dans certains cas, le temps de calcul du graph est linéaire par apport à sa taille.

Dans [I.S01], une mise en œuvre d'un algorithme de construction distribuée de l'espace d'état a été proposé. Cette algorithme a été intégré dans la boite à outils de vérification CADP : La fonction de partitionnement des états étant indépendente de leurs propriétés structurelles. De ce fait, cette fonction fait abstraction du langage de spécification utilisé.

L'un des algorithmes de modèle-checking qui implémente un algorithme de partitionnement dynamique est l'algorithme symbolique distribué proposé dans [GS00]. L'équilibrage de la mémoire est assuré par le repartitionnement de l'espace d'état à chaque fois que la mémoire est déséquilibrée. L'inconvénient de cette approche étant l'arrêt de la génération du graphe pendant l'étape de répartinnement. Par conséquent le temps de génération augmente considérablement.

Dans [O.G98], une approche de model-checking de logiciels modulaires a été développé. Un module dans le modèle correspond à un module du logiciel. Ce model checking se base sur la notion de supposition (en anglais "*assumption*") définie pour la représentation partielle de connaissances relatives aux valeurs de vérité de formules.

Dans [K.Y02], le problème de la réalisation d'un CTL model-checking distribué est fait en partitionnant l'espace d'états sur plusieurs nœuds. Chaque nœud impliqué dans le calcul distribué possède un espace d'état partiel et exécute un algorithme de model-checking sur cette structure incomplète. Pour être capable de procéder, les états de bord (feuilles du graphe) sont augmentés par des suppositions sur la vérité des formules. Les échangent entre eux les suppositions des d'états pertinents pour calculer plus d'informations précises. Les états de bord sont ces états qui appartiennent à d'autres nœuds du réseau. Chaque fois que

l'algorithme du modèle-checker atteint un état de bord il utilise l'information fournie par les autres nœuds du réseau sur la vérité des formules dans cet état. La connaissance partielle des suppositions nécessite recalculer assez fréquemment les nouvelles valeurs des suppositions.

Dans [Bou05a], le problème de la réalisation d'un CTL model checking distribué a été proposé. Par ailleurs, l'auteur présente un algorithme pour la distribution du modèle. Cette distribution prend en considération la connexité entre les états afin d'améliorer les performances de l'algorithme de vérification. La différence existant entre ce travail est le notre réside principalement dans la méthode de distribution du modèle. En effet la méthode proposée dans [Bou05b] suppose l'existence du modèle et procède à sa distribution. Cependant, la méthode que nous proposons effectue la distribution à la volet, ce qui nous épargne sa génération préalable précédent sa distribution.

4.3 Génération et distribution de l'espace d'états

Etant donné une spécification de haut niveau d'un système. La première étape du processus de vérification par model-checking consiste en la construction l'espace d'états à partir de cette spécification. 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. Afin de construire rapidement l'espace d'états du système à vérifier, nous distribuons la tâche d'exploration sur plusieurs machines, que nous appelons communément nœuds de calcul ou réseau de stations. Dans ce cas, chaque machine est responsable pour la génération d'une partie de l'ensemble des états du système. La génération distribuée est alors effectuée par plusieurs machines. Dans une telle opération, chaque machine génère une partie de l'espace d'états en calculant les successeurs des états qu'elle détient. Généralement, une fonction d'attribution appelée aussi fonction de hachage est utilisée pour distribuer l'ensemble des états entre les différentes machines. Une machine peut envoyer alors, selon la fonction d'attribution, un état à une autre machine. L'opération est initiée par la machine qui détient l'état initial.

4.3.1 Algorithme de génération distribuée

Dans ce paragraphe, nous présentons un algorithme pour la génération de l'espace d'états, ce dernier est inspiré de [I.S01]. On considère un ensemble de N machines : $W_0; W_1; W_2, ...W_{n-1}$ et un système représenté par un modèle M . Soit s_0 son état initial et $Succ$ une fonction de succession permettant de calculer, pour un état quelconque s : l'ensemble de ces états successeurs. L'idée de l'algorithme de génération est relativement simple. Il s'agit de distribuer la charge de calcul des états successeurs entre les différentes machines. Chaque machine W_i explore un ensemble d'états S_i. Elle construit alors une partie $\{S_i, U_i\}$ du graphe d'états. L'ensemble S_i est calculé à partir d'une fonction d'attribution d'états aux machines : h :

$S \longrightarrow [0..N-1]$ (S est l'ensemble de tous les états du système). Pour le stockage des états, chaque machine W_i stocke dans sa mémoire locale les informations concernant les états et les transtions de sa partie $\{S_i, U_i\}$.

Un états $s_i \in S_i$ ssi $h(s_i) = i, i \in [1..N-1]$

Un arc $u_i \in U_i$ ssi $u_i = (s, s\prime)$ et ($s \in S_i$ ou $s\prime \in S_i$).

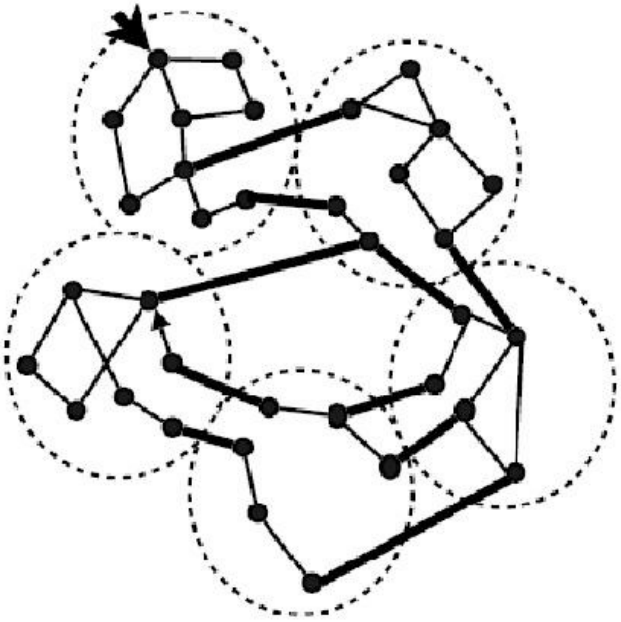
Les parties $\{S_i, U_i\}$ vérifient :

- $S = U_{i=0}^{N-1} S_i$ et $\forall$ 0 $\leq$ i, j $\leq$ N-1 , $S_i \cap S_J = \varnothing$ (L'ensemble des états est partitionné entre les N machines).

- $\sum_{i=0}^{N-1} U_i = U$ et $U_i \subseteq S_i \times S$ $\forall$ 0 $\leq$ i $\leq$ N-1 (répartition de l'ensemble des transitions).

-$\exists$ i vérifiant $0 \leq i \leq N-1$ tel que $s_0 \in S_i$.

La figure ci-dessous représente un espace d'états réparti sur plusieurs machines. Les états détenus par chaque machine sont dans le même cercle. Les transitions inter-machines sont en trait gras.



Représentation d'un espace d'états réparti sur plusieurs machines

Les états visités et explorés, par une machine W_i, sont stockés dans des ensembles disjoints : V_i (états visités non encore explorés) et E_i (états explorés dont les états successeurs n'ont pas encore été générés). Une machine W_i peut envoyer et recevoir des messages par invocation des primitives de communication. Les messages échangés sont de deux types.

- Le premier type englobe les messages de données utilisés pour échanger les états et les transitions entre les machines.

- Le deuxième type englobe les messages de contrôle inhérents à l'accomplissement de l'algorithme.

Le calcul est initié par une machine initiatrice. Cette machine doit avoir un index qui correspond *à* $h(s_0)$. Elle explore alors l'état initial s_0.
L'algorithme est composé d'une boucle principale, qui assure les différentes actions permettant de calculer tous les états atteignables.

L'algorithme distribué d'exploration est le suivant :

Algorithme 4.1 ***Distribution sur la machine*** W_i

Debut
$\mathrm{V}_i \longleftarrow \phi$
$\mathrm{E}_i \longleftarrow \phi$
$term_i \longleftarrow faux$
Si h(s_0) = i **Alors**
$\mathrm{V}_i \longleftarrow \{s_0\}$
Finsi
Tant que $(\neg\ term_i)$ **Faire**
Si $\mathrm{V}_i \neq \phi$ **Alors**
$\mathrm{V}_i \longleftarrow \mathrm{V}_i - \{s\}$
$\mathrm{E}_i \longleftarrow \mathrm{E}_i \cup \{s\}$
Pour chaque s'$\in$ succ(s) **Faire**
Si (h(s') = i) **Alors**
$\mathrm{V}_i \longleftarrow \mathrm{V}_i \cup \{s\}$
Sinon
Envoyer (s') à h(s') ;
Finsi
Finpour
Finsi
Si (Recevoir (message)) **Alors**
Si (message.type = état) **Alors**
Si (s' $\longleftarrow$ état (message) $\notin$ E_i **Alors**
$\mathrm{V}_i \longleftarrow \mathrm{V}_i \cup \{s'\}$
Finsi

Sinon
Si $V_i = \phi$ **Alors** procedure de terminaison() ; **Finsi**
Finsi
Finsi
Si i = 0 et $V_i = \phi$ **Alors**
Init_terminaison() ;
Finsi
$S_i \longleftarrow E_i$
Fintanque
Fin

4.3.2 Choix de la fonction de partition

Afin d'augmenter les performances de l'algorithme, il est essentiel de prévoir un équilibrage de charge entre les différentes machines. Le choix de la fonction de répartition joue un rôle très important pour garantir que les parties de l'espace d'états soient plus ou moins égales en terme de nombre d'états, de répartir les charges de calcul et de minimiser les transitions inter machines. Il est à noter que le choix de la fonction de répartition dépend fortement du modèle à analyser, du formalisme de modélisation et des vitesses des machines. De ce fait, selon ces critères, la fonction de répartition doit être choisie avec une étude préalable de l'évolution du système modélisé.

4.3.3 Complexité et efficacité de l'algorithme

La complexité de l'algorithme présenté est linéaire, elle est fonction de la taille de l'espace d'états. Les performances des algorithmes distribués dépendent de la charge de calcul à effectuer dans chaque machine du réseau. Mais, contrairement aux algorithmes séquentiels et aux algorithmes parallèles sur mémoire partagée, les algorithmes distribués s'exécutent sur un réseau de stations. C'est pour cela que leurs performances dépendent fortement du coût de communication. Ce coût dépend du nombre de messages échangés entre les différentes machines du réseau. Dans notre cas, le nombre de messages échangés est égale au moins au nombre de transitions inter machines ajouté au nombre de messages nécessaires pour la détection de la terminaison. Ce dernier est au moins égale à deux fois le nombre de machines moins un $(2 * (N - 1))$. Dans ce qui suit, nous discutons les optimisations que l'on peut envisager pour rendre l'algorithme plus efficace.

4.3.4 Détection de la terminaison distribuée

Dans un système distribué où les communications entre processus se font par échange de messages, le problème de détection de la terminaison d'une application faisant intervenir cet processus coopératifs est loin d'être pas un problème trivial. En effet des processus inactifs

à un instant donné peuvent être réveillés suit à la réception de messages en transites. Dans notre cas, la terminaison de l'algorithme de génération de l'espace d'états est conditionnée par :

- Chaque machine a terminé l'exploration des états qu'elle détient ($V_i = \varnothing$).
- Le nombre de messages reçu est égale au nombre de messages envoyés (pas de messages en transite).

La détection de la terminaison consiste à s'assurer que ces deux conditions sont vérifiées conjointement. Ceci est réalisé selon le principe suivant :

Nous supposons que toutes les machines forment un anneau virtuel qui leur permet de communiquer. Chaque machine W_i communique avec sa voisine $W_{(i+1) \bmod N}$ (Figure 4.1). L'algorithme de détection de la terminaison est le suivant :

Variables utilisées

Jeton ;
Pour chaque machine W_i on a deux variables
messagesenvoyési, messagereçusi : entier initialisé à 0

Comportement de la machine initiatrice (W_i)

Lors de la terminaison de l'exploration des états ($V_i = \varnothing$) et pas de jeton en circulation

Envoyer (jeton,1,0,0) ;

Lors de la réception de (jeton,vague,messagesenvoyés,messagesreçus)

Si vague = 0 **alors** terminaison := (messagesenvoyés = messagesreçus)
Sinon Envoyer (jeton,0,messagesenvoyés,messagesreçus)

Comportement d'une machine non initiatrice W_i

A la réception d'un message de donnée

messagesreçusi := messagesreçusi + 1

Lors de l'envoie d'un message de donnée

messagesenvoyési := messagesenvoyési + 1

Lors de la réception de (jeton,vague,messagesenvoyés,messagesreçus)

Attendre ($V_i = \varnothing$) ; (α)
Envoyer (jeton,vague,messagesenvoyés+messagesenvoyési,messagesreçus+messagesreçusi)

Remarque 4.1 *Notons que l'envoie d'un message sur l'anneau à pour destinataire la machine suivante. Remarquons aussi que dans la ligne (α), l'attente ne concerne pas les messages de l'application. D'autre part, l'exécution des parties de code ce l'algorithme est atomique.*

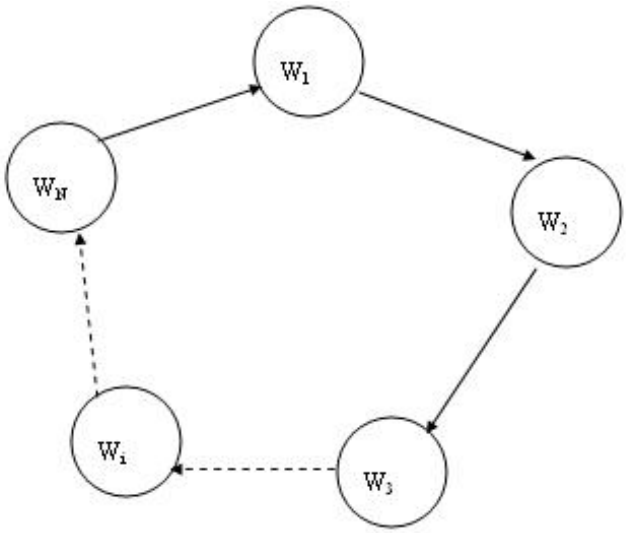


FIG. 4.1 – Anneau virtuel

L'exécution de la procédure de terminaison par une machine W_i est conditionnée l'absence d'états à explorer ($V_i = \varnothing$). Cette procédure de détection de la terminaison est différente de celle utilisée dans la version de génération distribuée de l'espace d'états parallèle de Spin [BS01] qui est complètement centralisé. L'avantage de cette procédure est que son principe permet de réduire le nombre de messages échangés entre les différentes machines [I.S01].

4.3.5 Parallélisme et équilibrage de charge

La plupart des algorithmes distribués de génération sont basés sur des procédures d'exploration (calcul d'accessibilité). Ces dernières nécessitent la transformation d'informations concernant les états et les transitions. Cependant, pour réduire le coût de communication, nous devons minimiser le nombre de transitions reliant deux états sur deux machines différentes (transitions inter machines -traits en gras dans la figure 4.3.1). En même temps, le nombre des états au niveau des machines doit être équilibré afin d'exploiter au maximum les possibilités du parallélisme. Le problème est de trouver une distribution qui réduit le nombre de transitions inter machines sans influencer l'équilibrage de charges et le temps moyen d'inactivité. Notons que la partition d'un graphe d'une façon équilibrée est très difficile. En effet, même en supposant que l'espace d'états est complètement généré, il n'est pas facile de trouver une distribution optimale. L'idée est d'essayer de trouver de bonnes valeurs pour les facteurs suivants :

Equilibrage de la charge : L'équilibrage de la charge peut être mesuré par l'écart type en considérant le nombre des états stockés sur chaque machine. Une autre manière de

mesurer ce paramètre consiste à considérer la notion du plus mauvais cas de charge définie comme suit :

$PMB = Max_{i=1..N} |((|S_i| - MOY)/MOY)|$.Tel que $MOY = (|S|/N)$ où N est le nombre de machines et S est le nombre de tous les états.

Taux des transitions inter machines (Cross transitions) : Ce facteur peut être mesuré à partir du rapport (nombre de transitions inter machines / nombre total de transitions)

4.3.6 Technique de la mémoire cache

Une deuxième technique que l'on peut utiliser pour améliorer les performances de l'algorithme est l'utilisation de la mémoire cache. Cette technique consiste à maintenir temporairement les états engendrés et envoyés à d'autres machines. Dans ce cas, seuls les états ne figurant pas dans la mémoire cache feront l'objet de transmission.

4.4 Vérification distribuée des propriétés

La génération distribuée de l'espace d'états a permis de répartir les états entre les différentes machines. Cela a permis de pallier au problème de l'explosion combinatoire de l'espace d'états. Une minimisation du temps de génération par rapport au cas séquentiel peut être atteinte pour certains systèmes par un bon choix de la fonction de répartition des états. Le but de cette génération distribuée est la vérification de systèmes de tailles importantes. De ce fait, les algorithmes de vérification feront aussi l'objet de distribution. Dans le reste de ce chapitre, nous exposons les différents points concernant la réalisation d'une vérification distribuée. En particulier, nous nous intéressons aux possibilités de distribuer les algorithmes de vérification des propriétés dites générales, des propriétés exprimées en logique temporelle linéaire et des propriétés exprimées en logique temporelle arborescente. Tout au long de cet exposé, nous supposons que la génération de l'espace d'états se déroule selon l'algorithme présenté dans la section précédente.

4.4.1 Vérification distribuée

Malgré que les travaux sur la distribution des espaces d'états ont pris la part du lion dans la littérature, nous commençons à voir de plus en plus de travaux sur la distribution des algorithmes de vérification ces dernières années [IB05] [K.Y02][BC03b][PLBJ04b][BJ03]. La vérification a concerné les propriétés de vivacité, de sûreté exprimées dans des logiques temporelles diverses (LTL, CTL, ...). Une classification de ces propriétés ainsi que des algorithmes pour leur vérification ont été agréblement étudiés dans [RAH03]. Dans ce qui suit nous synthétisons les résultats de cette étude. Ces propriétés peuvent être classées comme suit :

- Les propriétés dites générales ;
- Les propriétés décrites en logique temporelle linéaire ;
- Les propriétés décrites en logique temporelle arborescente.

4.4.2 Vérification distribuée des propriétés générales

Pour cette classe, nous nous intéressons à la vérification distribuée de deux types de propriétés :

1. Les propriétés sur les états, comme le blocage (deadlock) et les états non acceptés ou interdits (states reject)
2. Les propriétés sur les chemins, comme la vivacité du modèle, les séquences répétitives (cycles) et la réinitialisation (état d'accueil).

Nous notons que les propriétés du premier type peuvent être vérifiées à la volée. Dans ce cas, on ne génère qu'une partie de l'espace d'états et on stoppe la génération dès qu'on atteint une des situations précédemment citées. Ceci peut être utile même dans le cas des systèmes de tailles infinies [Sch]. Comme dans le cas séquentiel, les propriétés du deuxième type sont difficiles à vérifier, notamment la propriété de vivacité.

Vérification de la propriété de blocage

L'algorithme que l'on présente ici pour la vérification de la propriété de blocage est relativement simple. Il suffit de générer l'espace d'états et dès qu'on atteint un état qui n'a pas de successeurs, on stoppe la procédure de génération. Nous pouvons ainsi, afficher l'état de blocage et le chemin menant vers cet état. La procédure séquentielle de détection de blocage peut être facilement distribuée. En effet, dans le cas distribué, la détection des propriétés de blocage peut être réalisée de la façon suivante : Au moment de la génération distribuée de l'espace d'états, chaque machine teste l'ensemble des états successeurs des états qu'elle génère et dès qu'elle trouve qu'un ensemble de successeurs est vide, elle signale la présence d'un blocage et elle diffuse un message de terminaison aux autres machines pour arrêter l'exploration. L'algorithme distribué de détection des états de blocage est inclus dans l'algorithme distribué de génération de l'espace d'états. Cet algorithme peut être amélioré par l'ajout d'une variable qui permet la reprise de la génération de l'espace d'états après la détection d'un blocage. Comme il est important de l'améliorer par une procédure qui permet de différencier entre les états de terminaison et les états de blocage.

Algorithme 4.2 - *Détection distribuée de blocage*

Algorithme DDB (Machine i)
Si ($h(s_i) = i$) **Alors**
Si $succ(si) = \phi$ **Alors**
Reporter situation de blocage ;
SendAll(term_message) ;
Aller à fin ;
Sinon
// suite de l'algorithme de génération le cas où $h(s_i) = i$
finsi
sinon
// suite de l'algorithme de génération dans le cas où $h(s_i) \neq i$
Finsi
.....
Fin

Vérification distribuée de l'atteignabilité

Les propriétés d'atteignabilité sont habituellement les plus simples à vérifier. Quand on construit le graphe d'accessibilité d'un système donné, il est en principe possible de répondre à toutes les questions d'atteignabilité au moment de la construction. Nous nous limitons ici la présentation d'un exemple de propriétés d'atteignabilité qui concerne la détection distribuée des états non acceptés. Les autres propriétés d'atteignabilité peuvent être vérifiée de la même manière.

Exemple 4.1 *Détection des états de rejet (state reject) : Un état interdit est une situation que le système ne doit pas atteindre. Il est caractérisé par le fait qu'il ne satisfait pas une condition que doit le système vérifier. La détection des états interdits peut se faire durant la génération de l'espace d'états en testant chaque état généré. Dans le cas d'une exploration distribuée, chaque machine (station), teste successivement les états générés et si elle trouve un état interdit, elle envoie aux autres machines un message de terminaison. Le test des états interdits est considéré comme un cas particulier de la vérification d'accessibilité.*

Algorithme 4.3 - *Détection distribuée des états de rejet*

Algorithme DDER (Machine i)
Si ($h(s_i) = i$) **Alors**
Pour chaque (s_j dans $succ(s_i)$) **Faire**
Si interdit(s_j) **Alors**
Reporter s_j est interdit ;
SendAll(term_message) ;
Aller à fin ;
Sinon

// suite de l'algorithme de génération le cas où $h(s_i) = i$
finsi
sinon
// suite de l'algorithme de génération dans le cas où $h(s_i) \neq i$
Finsi
.....
Fin

Détection des cycles

Un cycle dans l'espace d'états représente une exécution répétée d'un ensemble d'actions (séquence répétitive). Intuitivement, un espace d'états (graphe d'états) contient un cycle si et seulement si un état est atteignable à partir de lui même. La détection des cycles dans le cas distribué est un peu compliquée par rapport au cas séquentiel. En effet, l'exploration en profondeur d'abord ne préserve pas l'ordre de génération dans le cas distribué à cause des vitesses différentes de calcul[BS01]. La détection des cycles au niveau du graphe d'états peut être traité comme le problème du model-checking LTL distribué[CG99] [BC03b][PLBJ04b]. Ceci sera détaillé dans le cadre de la vérification distribuée des propriétés décrites en logique temporelle linéaire.

Vérification distribuée de la vivacité

La vivacité est liée aux actions du système (les transitions). Nous rappelons que cette propriètè permet d'assurer que quelque soit l'évolution du système, on peut toujours aboutir *à* l'exécution d'une action donnée.
Par conséquent, nous avons besoin d'un espace d'états contenant toutes les informations concernant les actions exectuées par le système. Le même algorithme est utilisé pour la génération de l'espace d'états sauf qu'il est nécessaire de prévoir, pour chaque état, l'ensemble de ses successeurs, l'ensemble de ses prédécesseurs et les actions correspondantes.
Pour vérifier la propriété de vivacité, plusieurs algorithmes peuvent être utilisés, parmi lesquels, on trouve :

1. L'algorithme basé sur le marquage des ascendants et des descendants d'un sommet.
2. L'algorithme de tarjan.

L'idée consiste *à* calculer les composantes fortement connexes (CFC) du graphe représentant l'espace d'états. Par définition [Tar72], les CFCs sont des classes d'équivalence pour la relation d'accessibilité et de co-accessibilité : deux sommets u et v sont dans la même CFC si u est accessible depuis v et v est accessible depuis u .

On peut aussi remarquer que les CFCs peuvent se caractériser par une autre relation : depuis tous les sommets d'une même composante, on peut atteindre exactement le même ensemble de sommets. Cette caractérisation permet de se rendre compte de l'utilité du calcul de CFCs pour la vérification des propriétés d'accessibilité.
L'algorithme séquentiel de Tarjan [Tar72, Cou]permet de rechercher les composantes fortement connexes. Il se base sur une recherche en profondeur d'abord. Il a une complexité linéaire au nombre des arcs du graphe. L'algorithme de Tarjan est un algorithme de traversée, c'est *à* dire qu'il est intrinsèquement séquentiel[BC03b]. Le parallélisme est donc quasiment inexistant. Ceci mène *à* chercher d'autres algorithmes *à* paralléliser pour ce problème. Ils existent quelques algorithmes parallèles pour la détection des composantes fortement connexes qui ne se basent pas sur la recherche en profondeur d'abord. L'algorithme de Gazit et Miller [HG80] et ses extensions utilisent la multiplication des matrices pour localiser les CFCs.
Dans notre cas, l'algorithme présenté est inspiré de l'algorithme parallèle basé sur le marquage des ascendants et des descendants proposé en[LA00] .

Algorithme basé sur le marquage des ascendants et des descendants

L'idée principale de l'algorithme consiste *à* construire récursivement des partitions du graphe G= (V, E) qui coincident en fin de compte avec les composantes fortement connexes. Chaque itération de l'algorithme commence par le choix d'un état pivot x. Par la suite, l'algorithme cherche Descendants(x), l'ensemble des descendants de l'état x, qui sont tous les états accessibles *à* partir de x. similairement, il calcule Ascendants(x), l'ensemble des Ascendants de l'état x, qui sont tous les états depuis lesquels x est accessible.
Tous les états qui ne sont ni dans Descendants(x) ni dans Ascendants(x) sont classés dans Reste(x).Ce partitionnement se base sur le lemme suivant [LA00] :
Lemme 1 : La seule CFC contenant le sommet x de G=(V, E) est dans Descendants(x)∩ Ascendants(x). De plus, toute autre CFC est un sous-ensemble de Descendants(x), de Ascendants(x) ou sous-ensemble de Reste(x).
Basant sur ce lemme, le graphe est divisé en trois parties disjointes, et l'algorithme sera appliqué récursivement sur chaque partie. Le cas trivial est lorsque la partie contient un ou zéro état.
La complexité de la version séquentielle de cet algorithme est de l'ordre de $O(m\ log\ n)$ selon[LA00] pour un graphe qui a m sommets et n arcs. Dans ce qui suit, nous présentons une version distribuée de l'algorithme[RAH03].

Calcul des CFC -cas distribué

Considérons un graphe d'états G distribué sur plusieurs machines. On garde pour chaque état du graphe G, l'ensemble des numéros de ses successeurs et l'ensemble des numéros de ses prédécesseurs dans la mémoire locale de la machine qui le détient.

Notons :
Descendants (x) : l'ensemble des états accessibles *à* partir de x.
Ascendants (x) : l'ensemble des états depuis lesquels on peut atteindre x.
L'idée de l'algorithme est la suivante :
Si un état x n'appartient *à* aucune CFC, toutes les machines coopèrent pour calculer la CFC de x.
L'algorithme est le suivant :

Algorithme 4.4 ***-Calcul Distribué des CFCs***

Début CDCFCs (Machine i)
Tantque (non $term_i$) **Faire**
Si i est initiateur **Alors**
Si G est non vide **Alors**
selectionner un état pivot x (x n'appartient à accune CFC)
lancer Marquage Descendants(G,x) ;
lancer Marquage Ascendants(G,x) ;
CFC⟵ Descendants(G,x)∩Ascendants(G,x) ;
Finsi

Traitement des messages Reçu(Marquage et terminaison) ;
Si Terminaison(x) **Alors**
lancer le traitement de Descendants(G,x)-CFC ;
lancer le traitement de Ascendants(G,x)-CFC ;
lancer le traitement de Rest(G) ;
Finsi
Finsi
Fintantque
Fin

Pour la terminaison, c'est le même principe utilisé dans la génération distribuée de l'espace d'états. Nous pouvons améliorer cette version de l'algorithme par une procédure d'élimination des états n'appartenant à aucune CFCs. Cette procédure opère de la manière suivante :

Partant des états qui n'ont pas de prédécesseurs, on calcule une première liste des états. Ensuite, Partant des états sans successeurs, on calcule une deuxième liste.
Pour la première liste, nous procédons comme suit :
Chaque station met dans une file locale les états sans prédécesseurs. Puis, tant que la file n'est pas vide, on extrait un élément de la file et on le supprime du graphe. Puis, pour chaque successeur de l'élément tiré, on distingue deux cas :

1. Si l'élément successeur est locale, on décrémente le nombre de ses prédécesseurs. Si ce dernier devient zéro, on met l'élément successeur dans la file.

2. Si l'élément successeur est distant, on envoie à la station qui le détient un message qui permet à cette dernière de décrémenter le nombre de prédécesseurs de l'élément successeur, puis, le mettre dans la file locale si ce nombre devient zéro.
   L'élimination de la deuxième liste se fait de la même façon.

Vérification de la vivacité d'une transition

Une fois les CFCs déterminées, la vérification de la vivacité d'une transition est effectuée de la façon suivante :
Pour chaque CFC terminale (qui n'est source d'aucune transition), les machines coopèrent pour déterminer si une transition est vivante ou non. Chaque machine teste si la transition est possible à partir d'un état de cette CFC. Il est évident que la transition n'est vivante que si elle est vivante pour toutes les CFCs terminales [Mic]. Généralement, on n'a pas besoin toujours de tester toutes les CFCs pour toutes les machines.
Le principe du test distribué de la vivacité d'une transition est le suivant :
Nous considérons que les machines forment un anneau virtuel et que toutes les machines peuvent communiquer avec la machine initiatrice voir Figure 4.2.
La machine initiatrice initie la vérification en testant localement et pour chaque CFC la vivacité de la transition. Si la transition n'est pas vivante localement pour une CFC, elle envoie un message (Test-V, t, num-cfc) de test à la machine voisine. Toutes les autres machines testent périodiquement l'arrivée des messages tant qu'elles n'ont pas reçu le message *Stop*. Si la machine reçoit un message (Test-V, t, num-cfc), elle teste localement la vivacité de la transition. Si la transition est vivante, un message (t, num-cfc, True) est envoyé à la machine initiatrice. Sinon, un message (Test-V, t, num-cfc) est envoyé à la machine voisine, à l'exception de la machine de rang $N-1$ (N étant le nombre de machines) qui envoie un message (t, num-cfc, False) à la machine initiatrice.
Si la machine initiatrice reçoit un message (t, num-cfc, False), elle conclue que la transition n'est pas vivante et elle envoie un message (Stop) à toutes les machines. Nous pouvons conclure ainsi, que la transition t n'est pas vivante si la machine initiatrice reçoit un message (t, num-cfc, False).
Le traitement s'effectue en parallèle sur une sorte de pipe, le nombre de messages envoyés par chaque machine est égal au maximum au nombre de CFCs k. Le nombre de messages Stop est égal à N. Le nombre total de messages échangés sera donc au maximum $N*(k+1)$ qui ne se produisent que si la transition n'est pas vivante dans la dernière CFC testée.

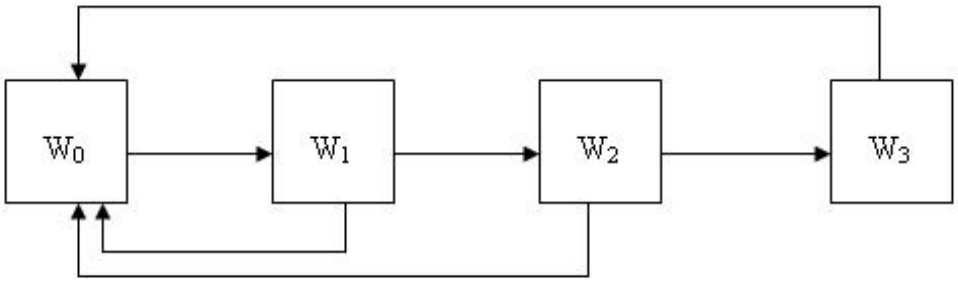


FIG. 4.2 – Exemple de l'architecture choisie avec 4 machines.

Vérification distribuée de la réinitialisation

La réinitialisation est un cas particulier de la vivacité. Un système est réinitialisable si le graphe d'états est fortement connexe. Le test de la réinitialisation du système se fait alors en calculant les composantes fortement connexes. Le système n'est réinitialisable que si le nombre des composantes est égal à 1. Comme dans la section précédente, le calcul des composantes fortement connexes consiste à attribuer à chaque sommet s du graphe un seul nombre c(s) et deux sommets s et s' font partie de la même composante si et seulement si $c(s) = c(s\prime)$ qui est vrai si s est accessible à partir de s' et que s' est à son tour accessible à partir de s.
Comme première étape pour la vérification, nous avons considéré les propriétés usuelles. Concernant la propriété de vivacité, l'algorithme que nous avons présenté n'est applicable qu'après la génération complète de l'espace d'états. La taille des systèmes que nous pouvons vérifier est fonction du nombre de machines et de leurs capacités mémoire. Il est à noter que le choix d'une bonne fonction d'attribution est parmi les points clés pour l'amélioration des performances des algorithmes présentés.
Dans ce qui suit, nous présentons les techniques de vérification des propriétés dynamiques de systèmes.

4.5 Le Model checking LTL distribué

4.5.1 Passage du model checking séquentiel au model checking distribué

Rappelons tout d'abord, que le model-checking LTL peut être réduit à la recherche des cycles acceptants dans le graphe d'un automate de Büchi. L'algorithme séquentiel du model-checking LTL est basé sur la recherche en profondeur (DFS). Un point important dans la recherche en profondeur (DFS) 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 pas préservé

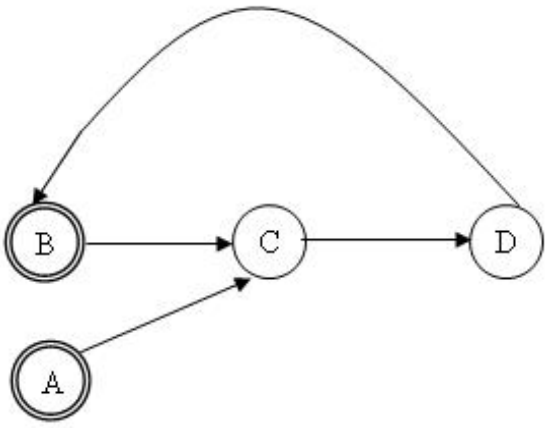


FIG. 4.3 – Exécution incorrecte de la procédure DFS dans le cas parallèle

à cause des différentes vitesses de calcul [BJ03]. Ce problème peut être illustré par le scénario suivant : Si deux processus Nested DFS s'exécutent simultanément au niveau des états A et B, le cycle passant par B peut ne pas être détecté (Fig 4.3). Le résultat dépend de la vitesse d'exécution des deux procédures Nested DFS. Ceci est le cas si le processus Nested DFS commençant à partir de l'état A visite l'état C avant le processus commençant à partir de l'ètat B. Ce dernier ne continuera pas l'exploration à travers l'état C. Par conséquent, le cycle BCDB ne sera jamais détecté(Fig. 4.3).

Dans l'exemple ci-dessus, nous avons montré que les procédures Nested DFS peuvent avoir des comportements incorrects dans le cas distribué.

D'une façon générale, lorsque les parties de l'espace d'états explorées par deux procédures Nested ont une intersection non vide, il y a une possibilité de ne pas détecter des cycles [BJ03]. Trois démarches peuvent être utilisées pour résoudre ce problème :

- Utiliser des structures de données supplémentaires pour stocker l'ordre global de DFS.
- Utiliser d'autres algorithmes qui ne nécessitent pas l'ordre de DFS.
- Distribuer l'espace d'états d'une manière à ne pas avoir des cycles partagés entre les différentes machines utilisées pour le calcul supplémentaire.

4.5.2 Nested DFS distribué avec une structure de données

Dans ce qui vient, nous allons décrire brièvement les trois démarches NESTED DFS distribuée avec structures de données supplémentaires. Cette approche utilise l'algorithme Nested DFS

[Hol03]. Elle permet une exécution parallèle de l'algorithme par plusieurs machines. Les états sont stockés aléatoirement au niveau des mémoires locales des différentes machines utilisées. L'algorithme Nested DFS est composé de deux procédures de recherche. La première trouve les états d'acceptation et la deuxième assure la détection des cycles passant par ces états. L'algorithme pourra avoir un comportement incorrect dans le cas distribué à cause de la non préservation de l'ordre DFS [LS99]. Ceci est bien illustré dans l'exemple précédent 4.3. Afin d'assurer un comportement correct de l'algorithme la deuxième procédure doit traiter les états d'acceptation (cherche les cycles) en respectant le post-ordre défini par la première procédure.

Une solution proposée dans [LS99] consiste à utiliser une structure de donnée supplémentaire afin de préserver l'ordre propre des états d'acceptation. La structure de donnée est construite de manière à garantir que la deuxième procédure (Nested) ne peut commencer le traitement d'un état que lorsque tous les états d'acceptation qui le précédent aient terminé leur deuxième procédure (Nested). En d'autre terme, cette structure de donnée conserve l'ordre entre les états. Parmi les model-checkers distribués opérant sur la logique LTL, on trouve la version distribuée de Spin [LS99]. Cet outil implémente un algorithme distribué nécessitant une exécution séquentielle de l'algorithme Nested DFS.

4.5.3 Détection des cycles négatifs

Le problème de détection des cycles acceptés peut être réduit au problème de détection des cycles ayant des coûts négatifs. La relation avec les automates de Büchi est la suivante : Un automate de Büchi correspond à un graphe dans lequel les transitions sont formées par des paires d'états. Si nous assignons des coûts aux transitions de manière à ce que les transitions qui partent d'un état d'acceptation sont initialisées à -1, et toutes les autres sont initialisées à 0. De ce fait les cycles négatifs coïncident avec les cycles d'acceptation. Ceci permet de réduire le problème du non vide de l'automate de Büchi de détection de cycles négatifs. Ce dernier problème est complètement lié au problème du plus court chemin de l'état source (SSSP : Single Source Shortest Path). Il s'agit de trouver les chemins menant d'un état source spécifique vers tous les autres états. La méthode séquentielle permettant de résoudre ce problème est présentée dans [CG99]. Elle calcule un graphe Gp qui maintient annoté par la distance d(s) pour chaque état. d(s) mesure la distance entre l'état s et son père P(s). Cette méthode doit être combinée avec une stratégie de détection de cycles. Les stratégies les plus connues sont : *walk to root*, *subtree traversal* et *subtree disassembly*. Une discussion concernant toutes ces stratégies peut être trouvée dans [BH03].

4.5.4 Détection des cycles basée sur le nombre des prédécesseurs

L'algorithme présenté dans [PLBJ04a] rentre dans le cadre de la deuxième approche. Il teste le vide d'un automate de Büch en se basant sur le calcul des prédécesseurs acceptés pour tester l'existence de cycles. Les états du graphe sont supposés numérotés. L'idée de l'algorithme

consiste à calculer le nombre de prédécesseurs acceptés pour chacun des états, appliquer une transformation sur l'ensemble des états et répéter l'opération jusqu'à l'obtention d'un point fixe [PLBJ04a].

4.5.5 Détection des cycles basée sur les transitions de retour en arrière

L'algorithme proposé en [BC03a] repose sur l'algorithme BFS. Il sert à détecter les cycles acceptants dans le graphe d'états. L'idée de cet algorithme distribué consiste à détecter tous les retours arrière par un algorithme BFS, puis détecter les cycles par un algorithme DFS parallèle en testant chaque retour en arrière.

4.5.6 Utililisation des composantes fortement connexes

A fin d'avoir un algorithme efficace pour la détection des cycles acceptants dans le cas distribué, il est intéressant de partitionner le graphe d'états d'une façon à ne pas avoir des cycles acceptés répartis entre les différentes machines. L'algorithme doit aussi permettre de limiter l'exécution du Nested DFS aux chemins qui peuvent réellement former des cycles. En revanche, il ne doit pas autoriser la visite d'un état par deux Nested DFS exécutés sur deux machines différentes. L'application de cette approche au model-checking LTL consiste à décomposer l'automate, représentant les comportements éventuellement erronés, en ses composantes fortement connexes. C'est l'idée de base de l'algorithme proposé dans [Bar02]. L'algorithme repose sur le fait qu'on peut assurer la validité de l'algorithme Nested DFS en partitionnant l'espace d'états d'une manière qui préserve la connexité des cycles du graphe représentant le produit synchronisé et en limitant la partie Nested de l'algorithme aux successeurs locaux. Cet algorithme n'est applicable que si le graphe est entièrement généré et réparti selon ses composantes fortement connexes sur l'ensemble des machines. Nous avons vu comment le model-checking LTL peut être distribué et adapté pour vérifier des espaces d'états répartis sur plusieurs machines. Nous avons remarqué que la distribution de sa version séquentielle n'est été pas triviale.

4.6 Model Checking CTL distribué

Comme pour le model-checking LTL, le model-checking CTL peut être distribué en adaptant sa version séquentielle au cas distribué. Toutes les machines contribuent alors pour la vérification des propriétés exprimées en CTL. Dans le cas distribué, les états sont stockés selon la fonction d'attribution des états aux machines. Les machines coopèrent alors afin de vérifier la formule CTL. L'adaptation du model checking CTL au cas distribué a été développé dans [RAH03]. Une version simplifiée de cet l'algorithme de model checking distribué est présentée dans ce qui suit.

4.6.1 Vérification des formule CTL simples

Les propositions atomiques et les expressions du premier ordre peuvent être vérifiées sans trôp de difficultés. La vérification de telles formules ne nécessite pas de communications entre les différentes machines. Chaque machine vérifie la satisfiabilité de la formule CTL simple dans la partie de l'espace d'états qu'elle détient.

Algorithme 4.5 ***Calcul distribué de sat***$(\phi = Pe)$

(Pe : Proposition élémentaire pour CTL) (Machine i)
$\mathrm{Sat}_i(\phi) \longleftarrow$vide
Pour chaque s dans S_i **Faire**
Si s vérifie Pe **Alors**
$\mathrm{Sat}(\phi) \longleftarrow \mathrm{Sat}(\phi) \cup \{s\}$
Finsi
Finpour

4.6.2 Vérification des formules de la forme $\Phi = EXf$

Soit une formule $f = EXf$, notons sat(ϕ) l'ensemble des états qui satisfont ϕ, l'ensemble que l'on veut calculer. Nous supposons sat(f) connue. L'algorithme séquentiel est présenté dans le Chapitre 3. Notons $\mathrm{sat}_i(f)$ l'ensemble des états qui satisfont f au niveau de la machine i . Une formule de la forme EXf peut être vérifiée dans le cas d'un graphe distribué de deux façons différentes :

Cas d'un chaînage avant -Backword computation

Dans le cas d'un chaînage avant, on parcourt l'espace d'états à partir de l'état initial. L'algorithme de calcul teste pour chaque état, si un de ses successeurs satisfait f . Si c'est le cas, l'état s satisfait ϕ. Le problème dans le cas d'un graphe distribué est que les successeurs d'un états s peuvent ne pas être locaux (s et ses successeurs ne sont pas au niveau de la même machine). Pour cela, dans un premier temps, on teste s'il y a parmi les successeurs locaux un état qui satisfait f. Dans ce cas, s satisfait f et le calcul se termine pour l'état s. Dans un deuxième temps, s'il n'y a aucun successeur local qui satisfait f, la machine sur laquelle se trouve l'état s doit envoyer des messages aux autres machines pour demander une réponse concernons la statifaction de f pour chacun des succeseurs non locaux . Si une machine répond positivement, l'état$s$ satisfait ϕ.

Algorithme 4.6 ***Calcul distribué de sat***$(\phi = EXf)$ ***(Machine i)***

$\mathrm{Sat}_i(\phi) \longleftarrow$vide
$\mathrm{Sat}_i(f)$ est connu ; Fin $\longleftarrow$ Faux ;

Pour chaque s dans S_i **Faire**
Pour chaque s' dans succ(s) **Faire**
Si s' est local et s' dans $Sat_i(f)$ **Alors**
$Sat_i(\phi) \longleftarrow Sat_i(\phi) \cup \{s\}$
Fin $\longleftarrow$ Vrai ;
Aller a Fin
Sinon
non_locaux $\longleftarrow$non_locaux $\cup\{s'\}$;

Finsi
Finpour
Si non Fin **Alors**
Pour chaque s' $\in$ non_locaux **Faire**
Envoyer (h(s'), [if_sat,s'],i) ;
Finpour
Tantque (non Fin) **Faire**
Recevoir (h(s'),[rep_if_sat],i) ;
Si rep_if_sat **Alors**

$Sat_i(\phi) \longleftarrow Sat_i(\phi) \cup \{s\}$
Fin $\longleftarrow$ Vrai ;
Aller a Fin
Fintanque
Finsi
Finpour
Fin

Cas d'un chaînage arrière (Forward computation)

Dans ce cas, on a besoin de stocker les prédécesseurs des états au moment de la construction du graphe. L'idée est de pouvoir calculer l'ensemble des états qui permettent d'atteindre des états cibles. Dans ce cas, l'algorithme distribué utilisé pour le calcul de sat (ϕ) est le suivant :

Algorithme 4.7 ***Calcul distribué de sat***$(\phi = EXf)$ ***(Machine i)***

$Sat_i(\phi) \longleftarrow$vide
$Sat_i(f)$ est connu ;
Pour chaque s' dans $Sat_i(\phi)$ **Faire**
Pour chaque prédécesseur s de s' **Faire**
Si s' est local **Alors**

$\text{Sat}_i(\phi) \longleftarrow \text{Sat}_i(\phi) \cup \{s\}$

Sinon

Envoyer (h(s), [mess_sat,s]) ;

Finsi

Finpour

Finpour

Fin

A la réception d'un message de type mess_sat la machine doit rajouter s à l'ensemble $\text{Sat}_i(\phi)$. Chacun des deux algorithmes précédemment exposés possède des avantages et des inconvénients. Dans le premier algorithme, on a pas besoins de stocker les prédécesseurs des états, mais, on a à parcourir l'ensemble des états fréquement en avant et en arrière et l'envoie de deux messages (question, réponse) pour savoir si un état satisfait la formule. Le deuxième algorithme est orienté, puisqu'il commence à partir de l'ensemble des états qui satisfont f, mais, nécessite le stockage des états prédécesseurs. Dans les algorithmes qui suivent, nous supposons que les états prédécesseurs sont stockés au moment de la construction du graphe d'états.

4.6.3 Vérification des formules de la formes $\Phi = E(\Phi_1 U \Phi_2)$

Pour évaluer une formule CTL de la forme EU, dans le cas distribué, toutes les stations opèrent dans leurs parties d'espace d'états. On procède exactement comme le cas de EX. Donc, si les prédécesseurs d'un état ne se trouvent pas au niveau de la même station, cette dernière doit envoyer les informations nécessaires pour évaluer les états prédécesseurs. Cet algorithme distribué est le suivant :

Algorithme 4.8 ***Calcule distribué de sat*** $\phi = E(\phi_1 U \phi_2)$ ***(Machine i)***

Debut

$\text{Sat}_i(\phi) \longleftarrow \text{Sat}_i(\phi_1)$;

$\text{Sat}'_i(\phi) \longleftarrow \text{Sat}_i(\phi_2)$;

$\text{Snew}_i \longleftarrow$ vide ;

Tant que $\text{Sat}'_i(\phi)$ n'est pas vide **Faire**

Pour tous les s dans $\text{Sat}'_i(\phi)$ **Faire**

Pour tous $s\prime$ dans Pred(s) **Faire**

Si $s\prime$ est local **Alors**

Si ((s' n'est pas $\text{Sat}_i(\phi)$ et (s' est dans $\text{Sat}_i(\phi_1)$)) **Alors**

$\text{Snew}_i \longleftarrow \text{Snew}_i \cup \{s'\}$;

$\text{Sat}_i(\phi) = \text{Sat}_i(\phi) \cup \{s'\}$

Finsi

Sinon
Envoyer (h(s), [sat_EU,s'],i) ;
Finsi
Finpour
Sat$'_i(\phi)$ ⟵Snew$_i$;Snew$_i$ ⟵vide ;
Finpour
Fin tant que
Fin

En cas de réception d'un message de type sat_EU par la machine i, elle doit rajouter $s\prime$ à sat$_i$ (ϕ) et à Snew$_i$

4.6.4 Vérification des formules de la formes $\Phi = A(\Phi_1 U \Phi_2)$

La distribution de la vérification concernant la formule for-all-until est aussi basée sur la version séquentielle, sauf que chaque machine traite sa partie de l'espace d'états. Chaque machine aura alors ses propres ensembles sat$_i(\phi)$, sat$'_i(\phi)$, sat$_i(\phi_1)$ et sat$_i(\phi_2)$. La détection de la terminaison est faite en envoyant un jeton de test. Comme il est noté ci-dessus, parfois il est impossible d'avoir tous les prédécesseurs d'un état donné dans la même machine. Cependant, pour un état s, on a à distinguer deux types de prédécesseurs :

1. Prédécesseurs locaux : Ce sont tous les prédécesseurs $s\prime$ de s, tel que $h(s) = h(s\prime)$
2. Prédécesseurs distants : Ce sont tous les précesseurs $s\prime$de s, tel que $h(s) \neq h(s\prime)$

Pour les prédécesseurs locaux on vérifie que chaque $s\prime$ satisfait ϕ, si c'est le cas, on teste pour chaque $s\prime$:
Si tous ses successeurs sont dans sat$_i(\phi)$? Ceci est réalisé en deux étapes :

1. Vérification de l'appartenance des successeurs locaux à l'ensemble sat$_i(\phi)$.
2. Dans le cas d'un successeur distant, lancement d'une vérification distribuée dist-succc check.

   Il est à noter que le test distribué n'est réalisé qu'après tous les tests locaux. L'algorithme de vérification dans le cas d'une formule $\phi = A(\phi_1 U \phi_2)$ est le suivant :

   Algorithme 4.9 ***Calcul distribué de sat*** $\phi = A(\phi_1 U \phi_2)$ ***(Machine i)***

Debut
Sat$'_i(\phi)$ ⟵Sat$_i(\phi_2)$;

Tant que (non Fin) **Faire**

Pour chaque s dans $\mathrm{Sat}_i(\phi)$ **Faire**
Pour chaque predecesseur $s\prime$ de s **Faire**
Si h(s) = i **Alors**
Si ((s' n'est pas $\mathrm{Sat}_i(\phi)$ et (s' est dans $\mathrm{Sat}_i(\phi_1)$) **Alors**
Si locat_succ(s') est inclus dans $\mathrm{Sat}_i(\phi)$ **Alors**
Si remote-succ(s') = vide **Alors**
$\mathrm{Sat}_i(\phi) \longleftarrow \mathrm{Sat}_i(\phi) \cup \{s'\}$

Sinon
dist-succ-check(s')
Finsi
Finsi
Finsi
Envoyer (h(s), s',msg_checkAU) ;
Finsi
Finpour
Finpour
Fin tant que
Fin

Les prédécesseurs distants sont envoyés aux machines correspondantes comme message de type msg-check-AU. Chaque machine réagit à l'arrivée des messages. A la réception d'un message de type msg-check-AU, la machine exécute la procédure distr-succ-check. Cette procédure est similaire au traitement d'un prédécesseur local. La procédure exécutée à la réception d'un message de type msg-checkAU se présente comme suit :

Algorithme 4.10 ***(Machine i)***

Debut
Pour chaque s' reçu **Faire**
Si ((s' n'est pas $\mathrm{Sat}_i(\phi)$ et (s' est dans $\mathrm{Sat}_i(\phi_1)$) **Alors**
Si locat_succ(s') est inclus dans $\mathrm{Sat}_i(\phi)$ **Alors**
Si remote-succ(s') = vide **Alors**
$\mathrm{Sat}_i(\phi) \longleftarrow \mathrm{Sat}_i(\phi) \cup \{s'\}$

Sinon
dist-succ-check(s')
Finsi
Finsi
Finsi
Finpour
Fin

Pour la vérification sur les successeurs, la machine i envoie un message à toutes les machines j sur lesquelles se trouvent les successeurs de $s\prime$ pour savoir si ces successeurs sont des éléments de $\mathrm{Sat}_j\ (\phi)$. Elle construit au même temps une tables de questions contenant des entrées $(s\prime, s_1, W_j)$, s_1 étant un sucesseur de $s\prime$. Chaque machine j répond par Vrai ou Faux. Lorsque la machine i reçoit une réponse Vrai, elle supprime la question correspondante de la liste des questions. Lorsque toutes les questions concernant l'états $s\prime$ sont supprimées, $s\prime$ doit être ajouté à $\mathrm{Sat}_i\ (\phi)$. Si la machine i reçoit la réponse FAUX, toutes les questions concernant l'état $s\prime$ peuvent être supprimées de la liste des questions. L'algorithme de la procédure distr-succ-check(s') est le suivant :

1. **Algorithme 4.11** ***(Machine i)***

Pour chaque s1 dans remote_succ(s') **Faire**

Quest [s', s1, h(s1)] $\longleftarrow -1$
Envoyer(h(s1),myid,msg-Q-check-succ,s1) ;
Finpour

lorsque une machine reçoi un message de type msg-Q-check-succ elle exécute la procedure suivante :

Axiom 1 ***Algorithme 4.12 (Machine i)***

Debut
Pour chaque s1 à recevoir comme msg-Q-check-succ **Faire**
recevoir(h(s1),remote_machine,msg-Q-check-succ,s1) ;
Si s1 est dans $\mathrm{Sat}_i(\phi)$ **Alors**
Envoyer(remote_machine,True,s1) ;
Sinon
Envoyer(remote_machine,False,s1) ;

Finsi
Finpour
Fin

Enfin, lorsqu'une machine reçoi un message de type True ou False elle réagit comme suit :

Algorithme 4.13 ***(Machine i)***

Debut
Si Recevoir(loc-machine,remote_machine, False, S1) **Alors**
supprimé toute entré de type Quest[s'] ;
Sinon
Si Recevoir(loc-machine,remote_machine, True, S1) **Alors**

Quest [s', s1, h(s1)] $\longleftarrow$ 1

Si toute entrée Quest[s']=1 **Alors**
$\text{Sat}_i(\phi) \longleftarrow \text{Sat}_i(\phi) \cup \{s'\}$

Finsi
Finsi
Finsi
Finpour
Fin

Remarque 4.2 *Afin d'améliorer les performances de la procédure dstr-check-succ, il est intéressant que la machine initiatrice de la procédure crée une seule question pour chaque machine au lieu d'une question pour chaque successeur, elle regroupe donc tous les successeurs d'une machine dans une liste pour les envoyer. Chaque machine questionnée teste alors si l'ensemble des successeurs font partie de son local Sat(ϕ) et elle envoie selon le résultat une seule réponse à la machine initiatrice. Ceci permet d'éviter des coûts supplémentaires en terme de communication. Ceci permet aussi d'éviter un nombre important de tests. Avec ces algorithmes, on peut vérifier sur des espaces d'états distribués toutes les formules CTL. La complexité de l'algorithme est linéaire à la taille du graphe et exponentielle à la taille de la formule.*

4.6.5 Conclusion

Dans ce chapitre, nous avons présenté les algorithmes les plus représentatifs dans la littérature pour la réalisation d'une vérification distribuée. Essentiellement, nous avons présenté des algorithmes distribués pour la vérification des propriétés générales, les approches utilisées pour la distribution du model checking LTL et un algorithme distribué pour la vérification des propriétés décrites en logique temporelle arborescente CTL. Dans le chapitre suivant, nous allons présenter un nouvel algorithme de vérification distribuée sur le formalisme des systèmes de transitions étiquetées maximales (STEM). Ceci permettra de positionner notre algorithme vis à vis des algorithmes présentés.

Troisième partie

Contributions

Chapitre 5

Construction parallèle réduite de l'espace d'états avec préservation de la α- équivalence

L'idée principale de la parallélisations de l'espace d'état dans la plupart des algorithmes distribués est la même [F.L99, I.S01] : le graph des états est distribué sur les nœuds du réseau, c'est-à-dire chaque nœud du réseau possède un sous-ensemble des états. Quand un nœud calcule un nouveau état, tout d'abord, elle vérifie (en utilisant la procédure Partition (s) ou h (s) dans notre cas) si l'état appartient à son propre sous-ensemble ou au sous-ensemble d'un autre nœud. si l'état est local, le nœud se poursuit au niveau local, sinon, un message contenant l'état est envoyé au nœud où cet état est présent. Nous supposons qu'il existe N nœuds disponibles. Chaque nœud a son propre processeur et une mémoire locale et peut communiquer avec d'autres nœuds par l'intermédiaire d'un réseau. Chaque nœud exécute une instance de l'algorithme parallèle, qui calcule un fragment de la STEM correspondant au système à vérifier .

5.1 Fonction de partition

Soit M un STEM et $h : S \rightarrow \{0, .., N-1\}$ est une fonction totale appelée une fonction de partition. Une partition de M en vertu de la fonction de partition h est un tuple $\pi_M^h = (M_0, .., M_{n-1})$, telle que $\forall i \in \{0..n-1\}, \quad S_i = (\{s \in S \mid h(s) = i\})$ .Cette fonction prend comme argument un état, représenté par une configuration, et retourne comme résultat l'identificateur (un entier qui représente le numéros du nœud) du nœud auquel il appartient. . Nous appelons $M_i$un fragment de M [K.Y02].

Dans notre implémentation nous avons considéré la fonction de partition h(s) = f(s ,P) mod N où P est un nombre premier et f(s ,P) est la fonction de hachage qui calcule le reste modulo P de la valeur de l'état s. La valeur de l'état s étant calculée à partir de sa

représentation en termes de configuration. La configuration étant codée par un entier long par l'utilisation de la fonction de cryptage MD5 (Message-Digest algorithme 5) [MD5]. La valeur ainsi obtenue est multipliée par une valeur constante. La valeur de P doit être soigneusement choisie pour que h distribue les états entre les N machines de manière uniforme. Dans le cas où le nombre de sites est premier (N premier), P peut être choisie égale à N. Par ailleurs, la fonction h de hachage fait correspondre le même indice de nœud i aux états, représentés par les configurations, qui sont alpha-équivalents. Cette dernière propriété est assurée de la manière suivante : Etant donnée une configuration G, premièrement en fait l'ordonnancement des événements maximales de cet configuration par la fonction sort, puis en fait la projection de la liste des événement maximale sur le premier élément par la fonction Proj, c'est-à-dire les actions , puisque nous avons représenter les événement maximales comme un couple (Action, Integer), ensuite en fait la traduction de cette configuration vers son expression de comportement LOTOS correspondante E par la fonction Config2Expr, en suite représente cette expression sous forme de chaîne de caractère par la fonction Expr2Str, enfin en fait la concaténation des actions de cet configuration avec l'expression E représenter sous forme de chaîne de caractère et on obtiens une nouvelle chaîne de caractère ; pour mieux comprendre voir l'exemple suivant :

Soit G = Stop [(a,1),(b,5),(a,3)] et

F = Stop [(b,3) , (a,2), (a,4)]

il est Claire de voir que c'est deux configuration sont alpha équivalent en appliquant les opérations suscite on a :

Proj∘Action2Str∘Sort([(a,1),(b,5),(a,3)]) = Sort(Action2Str (Proj(([(a,1),(b,5),(a,3)])) = Sort(Action2Str ([a,b,a])) = Sort("aba")

= "aab"

De meme pour F :

Proj∘Action2Str∘Sort([(b,3) , (a,2), (a,4)]) = Sort(Action2Str (Proj(([(b,3) , (a,2), (a,4)])) = Sort(Action2Str ([b,a,a])) = Sort("baa")

= "aab"

Expr2Str∘Config2Expr(G)=Expr2Str∘Config2Expr(F) = Expr2Str∘Config2Expr(Stop) = "Stop"

Concat ("Stop", "aab") = "Stopaab"

Ces opérations sont représentées dans notre implémentation par la fonction Haskell :

config2str :Config->String

Si deux configurations sont alphas équivalents la fonction Haskell config2str retourne la même chaîne de caractère, donc le même indice de nœud i, comme dans notre exemple cette fonction retourne la même chaîne "Stopaab", donc par l'application de la fonction de partition sur cette chaîne en obtiens le même indice de nœud. Notre fonction de partition est base sur la fonction de cryptographie MD5. Le module MD5 utilisé dans notre application est développé par [MD5].

5.2 Pourquoi la fonction MD5

Toute fonction qui prend une chaine de caractère, faites certain calcul sur cette chaine et retourne un nombre entier, pourrait être appropriée. Par exemple, la fonction qui rend la taille d'une chaine pourrait être acceptable, mais dans le cas du model checking parallèle nous devons ajouter trois autres conditions :

- La fonction doit accepte les caractères 8-bit et non seulement les caractères alphanumériques par exemple la fonction définie dans [Aho] laquelle est utilisée dans les compilateurs pour entreposer des identificateurs dans la table des symboles serait non approprié

- La fonction a une bonne dispersion pour éviter d'avoir trop de collisions (la fonction qui rend la taille serait non appropriée)

- La fonction a une bonne vitesse de calcul

pour toutes ces raison, la fonction MD5 est tres convenable pour la construction parallel d'espace d'etats.

5.2.1 Origine

MD5 (Message Digest 5) est une fonction de hachage cryptographique qui calcule, à partir d'un fichier numérique, son empreinte numérique (en l'occurrence une séquence de 128 bits ou 32 caractères en notation hexadécimale) avec une probabilité très forte que deux fichiers différents donnent deux empreintes différentes.

En 1991, Ronald Rivest améliore l'architecture de MD4 pour contrer des attaques potentielles qui seront confirmées plus tard par les travaux de Hans Dobbertin.

Cinq ans plus tard, en 1996, une faille qualifiée de grave (possibilité de créer des collisions à la demande) est découverte et indique que MD5 devrait être mis de côté au profit de fonctions plus robustes comme SHA-1.

En 2004, une équipe chinoise découvre des collisions complètes. MD5 n'est donc plus considéré comme sûr au sens cryptographique. On suggère maintenant d'utiliser plutôt des algorithmes tels que SHA-256, RIPEMD-160 ou Whirlpool.

Cependant, la fonction MD5 reste encore largement utilisée comme outil de vérification lors des téléchargements et l'utilisateur peut valider l'intégrité de la version téléchargée grâce à l'empreinte. Ceci peut se faire avec un programme comme md5sum pour MD5 et sha1sum pour SHA-1.

MD5 peut aussi être utilisé pour calculer l'empreinte d'un mot de passe ; c'est le système employé dans GNU/Linux avec la présence d'un sel compliquant le décryptage. Ainsi, plutôt que de stocker les mots de passe dans un fichier, ce sont leurs empreintes MD5 qui sont enregistrées, de sorte que quelqu'un qui lirait ce fichier ne pourra pas découvrir les mots de passe.

Le programme "*John the ripper*" permet de casser les MD5 triviaux par force brute. Des serveurs de « tables inverses » (à accès direct, et qui font parfois plusieurs gigaoctets) permettent de les craquer souvent en moins d'une seconde .

Voici l'empreinte (appelée abusivement signature) obtenue sur une phrase :
MD5("Wikipedia, l'encyclopedie libre et gratuite") = d6aa97d33d459ea3670056e737c99a3d
En modifiant un caractère, cette empreinte change radicalement :
MD5("Wikipedia, l'encyclopedie libre et gratuitE") = 5da8aa7126701c9840f99f8e9fa54976

5.3 Algorithme distribué de construction de l'espace d'états

L'algorithme 5.1 est une version simplifiée de la construction distribuée de l'espace d'états réduit modulo l'alpha équivalence.

Algorithme 5.1

Data : initial configuration G_ϕ
Variables :
L_i :the list of configurations not yet explored
E_i : the list of configurations explored
$Graph_i$: the list of transition
Begin
If h(G_ϕ) = i **then**
L_i.push(G_ϕ)
Endif
While (shutdown signal not received) **Do**
begin
While L_i is not empty **Do**
Begin

L_i.pop(G)
Exhaustive development /* SOS of Maximality */
For each new configuration G' **Do**
Begin
If h(G') = i **then**
Begin
- check if there exist G" in L_i or in E_i
where G' $=_\alpha$ G"
If G" exist **Then**
- make a list of substitution Sub

- substitute the target G' in the new
transition (G, atom , G') by G"
-make the necessary substitution of
events in L_i and in E_i
(a) $Graph_i := Graph_i$ + (G, atom , G")
else
L_i.push(G')
(b) $Graph_i := Graph_i$ + (G, atom , G')
Endif
else
(c) $Graph_i := Graph_i$ + (G, atom , G')
SEND(h(G'), G')
Endfor
E_i.push(G)
Endwhile
(d) **While** *RECEIVE*(G) **Do**
check if there exist G" in L_i or in E_i
where G $=_\alpha$ G"
If G" not exist **Then**
L_i.push(G)
Else
- make a list of substitution Sub
-make the necessary substitution of
events in L_i and in E_i
Endif
Endwhile
Endwhile
End.

Afin d'expliquer cet algorithme, considérons une autre fois l'exemple de l'expression de comportement LOTOS :

E = "a ; d ; stop | [*d*] | b, d ; stop"

pour simplifier la comprehension supposons que :

N = 3 (Supposons que le nombre N de nœuds est égal à 3),
h = (MD5(S) mod 67) mod 3.

Nous concentrons notre attention pour le moment sur les états (configurations seulement) et nous omettons les transitions. la configuration Initiale est :$({}_{\emptyset}[a, d; stop|[d]|b, d; stop])$

$L_i = []$ and
$E_i = []$

La liste non encore exploré et la liste explorée sont vides pour chaque nœud i. Dans notre implementation la configuration initiale $({}_{\emptyset}[a, d; stop \mid [d] \mid b, d; stop])$ est écrite sous la forme suivante :
$G_{\emptyset}$= ParallelConf (PrefixeConf []"a"(Prefixe "d" Stop))(PrefixeConf[]"b"(Prefixe "d" Stop))["d"].

la première étape (a) dans algorithm 1, consiste à déterminer l'indice du nœud initiateur du calcul. D'où la transformation de la configuration en une chaîne de caractères par l'utilisation de la fonction Conf2str ; On obtient alors : :

S = Conf2str (G) ="PrarallelConfPrefixeConfaPrefixedStopPrefixeConfbPrefixedStop[d]" ,
MD5(S)= "5b3a74cb6cc1aaec81f11301ad09100f" ,
h(s) = (MD5(s) mod 67) mod 3 = 2,

Nous obtenons ce résultat après le calcul de la valeur décimale équivalente à la valeur hexadécimale MD5 (S). De ce fait le nœud initiateur est celui de numéro 2 :
$L_2 = [({}_{\emptyset}[a, d; stop|[d]|b, d; stop])]$ and
$E_2 = []$.

- Dans la première itération (b) de l'algorithme par le nœud 2, à partir de la configuration initiale, nous pouvons générer à travers les règles de la sémantique opérationnelle de Maximalité deux configurations : $({}_{(a,1)}[d; stop]|[d]|\ {}_{\emptyset}[b, d; stop])$ et $({}_{\emptyset}[a, d; stop]|[d]|\ {}_{(b,1)}[d; stop])$
  où :
  $h(({}_{(a,1)}[d; stop]|[d]|_{\emptyset}[b, d; stop])) = 1$,
  $h(({}_{\emptyset}[a, d; stop]|[d]|_{(b,1)}[d; stop])) = 0$,

  Chaque configuration sera donc envoyée au nœud correspondant (nœud 1 et nœud 0). D'où :

  $E_2 = [{}_{\emptyset}([a, d; stop|[d]|b, d; stop])]$
  $L_2 = []$
  $L_1 = [({}_{\{(a,1)\}}[d; stop]|[d]|_{\emptyset}[b, d; stop])]$

$E_1 = []$
$L_0 = [(_{\emptyset}[a, d; stop]||[d]|_{\{(b,1)\}}[d; stop])]$
$E_0 = []$

Aucun cas d'équivalence n'est détecté dans (d) car toutes les listes sont vides dans les nœuds 1 et 0.

- le nœud numéros 1 commence le calcul maintenant et dans la première itération (b) , génère apartire de la configuration $(_{\{(a,1)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop])$ la configuration $(_{\{(a,1)\}}[d; stop]||[d]||_{\{(b,2)\}}[d; stop])$ où :
$E_1 = [_{\{(a,1)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop])]$
$h(_{\{(a,1)\}}[d; stop]||[d]|(b, 2)[d; stop]) = 1$
$L_1 = [(_{\{(a,1)\}}[d; stop]||[d]|_{\{(b,2)\}}[d; stop])]$
Dans ce cas aussi, aucun cas d'équivalence n'est apparu. De meme le nœud zéro aussi commence le calcul en parallèle avec le nœud numéros 1 et dans la première itération (b), génère apartire de la configuration $(_{\emptyset}[a, d; stop]||[d]|_{(b,1)}[d; stop])$ la configuration $(_{\{(a,2)\}}[d; stop]||[d]||_{\{(b,1)\}}[d; stop])$ où :

$h(_{\{(a,2)\}}[d; stop]||[d]|_{\{(b,1)\}}[d; stop]) = 1$

La configuration $(_{\{(a,2)\}}[d; stop]||[d]|(b, 1)[d; stop])$ est donc envoyée au nœud 1.

Au niveau du nœud 1, la ligne (d) de l'algorithme vérifie si la configuration reçue $(_{\{(a,2)\}}[d; stop]||[d]||_{\{(b,1)\}}[d; stop])$ est alpha équivalente à une configuration déjà existante dans L_1 ou dans E_1 :

$E_1 = [_{\{(a,1)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop])]$
$L_1 = [(\{(a, 1)\}[d; stop]||[d]|_{\{(b,2)\}}[d; stop])]$.

La configuration reçue étant alpha équivalente à $(_{\{(a,2)\}}[d; stop]||[d]||_{\{(b,1)\}}[d; stop])$. ce qui induit la fonction de substitution suivante :

Sub = (((a, 1), (a, 2)), ((b, 2), (b, 1))) où
$\{(a, 1), (a, 2)\}$ / (a, 3) and {(b, 2), (b, 1)} / (b, 3).

Intuitivement, les événements (a,1) et (a,2) seront substitués chacun par (a,3) et les événements (b,1) et (b,2) par (b,3). Le symbole /est utilisé pour représenter la substitution. On obtient donc après substitution :

$E_2 = [(_{\emptyset}[a, d; stop|[d]|b, d; stop])]$
$L_2 = []$
$E_1 = [(_{\{(a,3)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop])]$
$L_1 = [(_{\{(a,3)\}}[d; stop]||[d]|_{\{(b,3)\}}[d; stop])]$
$E_0 = [(_{\emptyset}[a, d; stop]||[d]|_{\{(b,3)\}}[d; stop])]$
$L_0 = []$

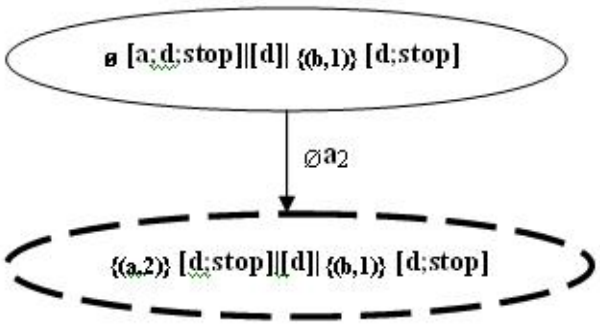
Etant donnés que les listes L_0 et L_2 sont vides, les nœuds 0 et 2 se mettent en attente de la réception de nouvelles configurations, alors que le nœud 1 poursuit le calcul. Pendant le traitement de la configuration $({}_{\{(a,3)\}}[d; stop]||[d]|_{\{(b,3)\}}[d; stop])$ à la deuxième itération (b), le nœud 1génère la configuration : $({}_{\{(d,4)\}}[stop]||[d]|_{\{(d,4)\}}[stop])$ et on obtient : $h({}_{\{(d,4)\}}[stop]||[d]|_{\{(d,4)\}}[stop]) = 1$, sans aucun cas d'équivalence à signaler. Cette configuration est rajoutée à L_1, d'où :

$E_2 = [(_{\emptyset}[a, d; stop|[d]|b, d; stop])]$
$L_2 = []$
$E_1 = [({}_{\{(a,3)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop]),$
$({}_{\{(a,3)\}}[d; stop]||[d]|_{\{(b,3)\}}[d; stop])]$
$L_1 = [(\{(d, 4)\}[stop]||[d]|_{\{(d,4)\}}[stop])]$
$E_0 = [(_{\emptyset}[a, d; stop]||[d]|_{\{(b,3)\}}[d; stop])]$
$L_0 = []$

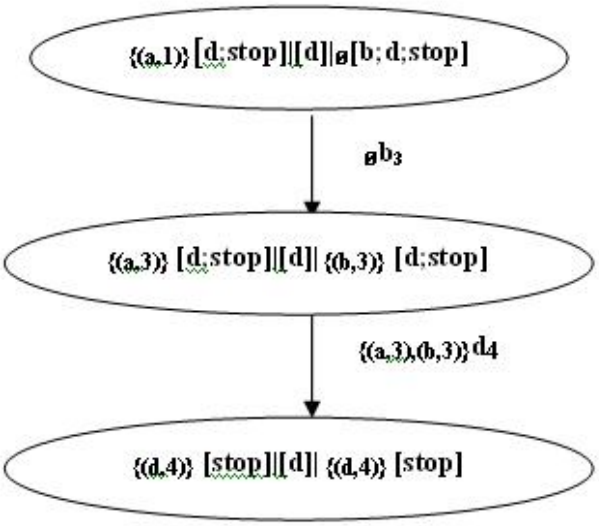
La troisième itération est le point d'arrêt de cet algorithme. En effet, la seule configuration à développer correspond à un état terminal. Le stem alpha réduit est le même que celui obtenu dans l'itération précédente. Toutes les listes non explorées deviennent vides.

$E_2 = [(_{\emptyset}[a, d; stop|[d]|b, d; stop])]$
$L_2 = []$
$E_1 = [({}_{\{(a,3)\}}[d; stop]||[d]|_{\emptyset}[b, d; stop]),$
$({}_{\{(a,3)\}}[d; stop]||[d]|_{\{(b,3)\}}[d; stop]), ({}_{\{(d,4)\}}$
$[stop]||[d]|_{\{(d,4)\}}[stop])]$
$L_1 = []$
$E_0 = [(_{\emptyset}[a, d; stop]||[d]|_{\{(b,3)\}}[d; stop])]$
$L_0 = []$

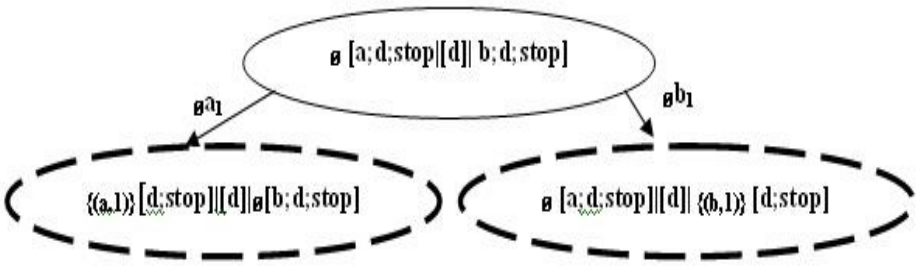
Le partitionnement du stem entre les nœuds 0, 1 et 2 est représenté par les figures 5.3, 5.3 et 5.3. Les États représentés par des cercles discontinus, ne sont pas présents dans le nœud. Ce sont des états qui appartiennent à l'ensemble des états border.



Fragment du stem dans le nœud 0



Fragment du stem dans le nœud 1



Fragment du stem dans le nœud 2

5.4 Cas des réseaux hétérogènes

L'algorithme de construction de l'espace d'états fonctionne convenablement tant que ce dernier s'exécute sur des machines identiques. Cependant un réseau est souvent composé de machines hétérogènes dont chacune a sa propre représentation des nombres, des caractères etc, et bien sur des états échangés entre les nœuds. A titre d'exemple, les gros systèmes IBM utilisent le code EBCDIC pour les caractères alors que les ordinateurs personnels IBM utilisent le code ASCII. En conséquence, il n'est pas possible d'envoyer un état sous forme de chaîne de caractères d'un nœud IBM PC vers un gros système IBM car ce dernier interpréterait le caractère de façon incorrecte. Des problèmes similaires apparaissent avec la représentation des entiers (complément à 1 ou à 2). Un autre problème provient du fait que certains processeurs, comme le 386 d'intel, numérotent leurs octets de la droite vers la gauche, alors que d'autres, comme le SPARC de Sun, les numérotent dans l'autre sens. Le format Intel est appelé *"little indian"* et le format du SPARC étant le "*big indian".* Pour résoudre ce problème dans le contexte des langages de programmation compilés on doit définir un protocole de communication tel que le XDR des services web pour avoir une représenlation commune. Dans notre cas d'implémentation, le problème ne se pose pas car nous avons utilisé un langage fonctionnel qui s'exécute sur une machine virtuelle tel que java.

5.5 Conclusion

L'objectif de ce travail est de présenter un algorithme distribué de génération des systèmes de transitions étiquetées maximales, avec prise en compte de la réduction modulo l'alpha

équivalence[ZEAB08],[ZEAB09]. En peut distinguer notre travail par rapport aux autres travaux par le fait que dans notre contexte en n'a pas besoin d'envoyer les transitions, ce qui réduit la charge de communication et augmente les performances. Ceci est dû essentiellement à deux facteurs, le premier facteur concerne le modèle sémantique du parallélisme utilisé, ce dernier est riche en informations associées aux états. Le deuxième facteur réside dans le fait que notre algorithme de vérification distribuée exploite le modèle sémantique de maximalité malgré l'insuffisance d'information causée par la répartition du graphe. Ce point sera développé dans le chapitre suivant. Par ailleurs notre algorithme n'exclut pas les techniques de réduction centralisées spécifiques au modèle sémantique de maximalité, on peut dire que l'originalité du travail vient de l'utilisation du modèle de maximalité en tant que modèle sémantique pour l'approche de vérification symbolique nommée model checking.

Chapitre 6

Vérification Distribuée basée sur la sémantique de maximalité

6.1 Système de Transitions étiquetées maximales en tant que structure de Kripke

Soit M=$(\Omega, A, \mu, \xi, \psi)$ un STEM tel que $\Omega = (S, T, \alpha, \beta, s_0)$,et soit $K = (S, L, R, I)$ une structure de kripke. Remarquons que la simplification d'un stem en omettantles anotations des transition et en considérant que chaque nom d'événement est une proposition atomique ; la structure résultante peut être vue comme une structure de Kripke. En d'autre terme le quadruplet (S, ψ, T, s_0) est une structure de Kripke.

Exemple 6.1 *: Nous prenons l'exemple du STEM de la Figure 2.1, la structure de Kripke qui lui est associée est représentée par la Figure 6.1.*

Nous faisons la remarque que le modèle des STEMs est très riche d'information et qu'il peut être utilisé pour le scheduling dans une plate-forme multiprocesseur. En effet, ce modèle

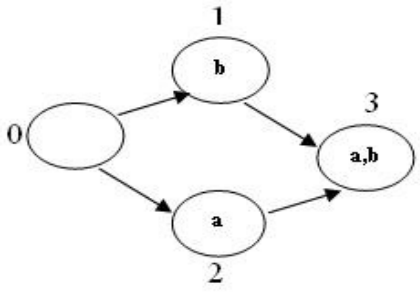


FIG. 6.1 – Structure de Kripke

intègre aussi bien les informations sur le parallélisme que celles des dépendences causales entre exécutions d'actions. Pour la vérification logique de propriétés comportementales nous pouvons donc se contenter des structures de Kripke associées aux STEMs. Ces derniers étant construits de manière opérationnelle.

Définition 6.1 *Soit $M = (S, L, R, I)$ une structure Kripke.*

- *L'ensemble des états terminaux de M (état sans successeurs) est $border(M) = \{s \in S \mid \neg\exists s'.(s, s') \in R\}$.*

- *Soit $T \subseteq S$. Nous définissons une structure de Kripke partiale $\mathcal{F}_M(T) = (S_T, L_T, R_T, I_T)$ comme suit :*

  $S_T = \{s \mid s \in T \text{ or } \exists s' \in T : (s', s) \in R\}$
  $R_T = \{(s_1, s_2) \in R \mid s_1 \in T, s_2 \in S_T\}$
  $I_T = \{s \in S_T \mid s \in I\}$
  $L_T : S_T \times P \to \{false, \bot, true\}$

Nous appelons la structure de Kripke partielle $\mathcal{F}_M(T)$ un *fragment* de M. Les états de l'ensemble T sont tous présents dans le nœud et $S_T - T$ est l'ensemble des états terminaux de $\mathcal{F}_M(T)$, c'est-à-dire $border(\mathcal{F}_M(T))$. A partir de la définition 6.1, nous pouvons constater que le fragment $\mathcal{F}_M(T)$ connaît tous les *successeurs immédiats* des états présents dans le nœud, c'est-à-dire en fonction de T. La fonction de vérité L_T pour les fragments de M et la formule CTL φ, est une fonction totale définit comme suit :
$L_T : S \times \varphi \to \{True, \bot, False\}$.
$L_T(s, \varphi) = True$ ssi $M, s \models \varphi$ et $L_T(s, \varphi) = False$ ssi $M, s \not\models \varphi$. Nous utilisons $L_T(s, \varphi) = \bot$ dans le cas où nous ne connaissons pas la valeur de vérité dans une certaines étape du calcul de la fonction de vérité. Entre autre au début du calcul de la vérité des états terminaux. La validité d'une formule vis à vis de la structure de Kripke complète M est une fonction totale $L : S \times \varphi \to \{True, False\}$. Puisque toutes les informations nécessaires pour le calcul sont disponibles, nous n'avons pas besoin de la troisième valeur représentée par le symbole d'incertitude $\bot$.

Proposition 6.1 *L'algorithme 2 de la construction parallèle de l'espace d'états construit des fragments :.*

Preuve. Soit T l'ensemble des état présents dans le nœud i. A partir de l'algorithme, nous pouvons voir que pour chaque nouvel état $G\prime$ calculé à partir d'un état G. Soit $G\prime$ est local,

c'est-à-dire :$h(G\prime) = i$,$G\prime \in T$, ou non, c'est-à-dire $h(G\prime) \neq i$ mais $(G, G\prime) \in R$. Il est plus facile de voir que tous ces états représentent S_T. A partir de la définition 6.1, nous pouvons comprendre que tous les états dans le nœud connaissent tous leurs successeurs immédiats, en d'autre terme, toutes les transitions d'un état présent dans le nœud sont présentes dans le même nœud. Ce qui est assuré par l'algorithme. ■

Définition 6.2 *Soit* $M = (S, L, R, I)$ *une structure de Kripke , et* $\mathcal{F}_M(Z)$ *un fragment de* M *, nous definissons*
$T(p) = \{s \in S \mid L(s,p) = true\}$
$U(p) = \{s \in S \mid L(s,p) = \perp\}$
$F(p) = \{s \in S \mid L(s,p) = false\}$

pour $s \in T, inT(s) = (1, s)$
pour $s \in U, inU(s) = (\perp, s)$
pour $s \in F, inF(s) = (o, s)$

$T^+(p) = \{inT(s) \mid \forall s \in T(p)\}$
$U^+(p) = \{inU(s) \mid \forall s \in U(p)\}$
$F^+(p) = \{inF(s) \mid \forall s \in F(p)\}$
$S^+(p) = T^+(p) \cup U^+(p) \cup F^+(p)$

$\forall e_1, e_2 \in S^+$ *nous avons* $e_1 \doteq e_2$ *ssi* $snd(e_1) = snd(e_2)$

Nous interprétons les opérateurs logiques $\wedge$ et $\vee$ sur une structures de Kripke partielle en utilisant la logiques de Kleene à trois valeurs. Une interprétation précise et compatible des connecteurs de Kleene a été donnée par Korner. Ce dernier a défini la notion de classe inexacte d'un domaine non vide donné A généré par une définition partielle $D(p)$ d'une propriété p des éléments de A comme une fonction caractéristique à trois valeur .

$X_p : A \rightarrow \{-1, 0, 1\}$

$$X_p(a) = \begin{cases} -1 \\ 0 \\ 1 \end{cases}$$

$X_P : A \rightarrow \{-1, 0, 1\}$
-1 ou p(a) en fonction de d(p) est fausse
XP (a) = 0 ou p(a) en fonction de d(p) est inconnue
1 ou p(a) en fonction de d(p) est vraie

Toute famille des classes inexactes d'un domaine A donné est un treillis de Morgan, les opérations algébriques $\cup$, $\cap$ et $-$ sont définies comme suit :

$(X \cup Y)(a) = max\{X(a), Y(a)\}$
$(X \cap Y)(a) = min\{X(a), Y(a)\}$
$(-X)(a) = -X(a)$

Ces opérateurs sont homologues aux connecteurs de Kleene. Nous allons maintenant examiner notre proposition basée sur la théorie mentionnées ci-dessous :

- Soit $\mathbb{B}_{\perp} = \{false, \perp, true\}$.
- Soit $(\mathbb{B}_{\perp}, <)$ un ordre total tel que $false < \perp < true$
- $\forall e_1, e_2 \in S^+$ dans le cas où $e_1 \doteq e_2$ nous avons :
  - $e_1 \wedge e_2 = (min(fst\ e_1, fst\ e_2), snd\ e_1)$
  - $e_1 \vee e_2 = (max(fst\ e_1, fst\ e_2), snd\ e_1)$
- Soit $S \subseteq S^+$ et $G \subseteq S^+$ nous définissons :
  - $S \sqcap G = \{e_1 \wedge e_2 \mid \forall e_1 \in S, \forall e_2 \in G\ tel\ aue\ e_1 \doteq e_2\}$
  - $S \sqcup G = \{e_1 \mid \forall e_1 \in S \quad et\ \nexists e_2 \in G\ tel\ que\ e_1 \doteq e_2\} \cup \{e_2 \mid \forall e_2 \in G \quad et\ \nexists e_1 \in S\ tel\ que\ e_1 \doteq e_2\} \cup \{e_1 \vee e_2 \mid \forall e_1 \in S, \exists e_2 \in G\ tel\ que\ e_1 \doteq e_2\}$
- Soit *succ* une fonction définie comme suit :
  - succ : $S^+ \rightarrow 2^{S^+}_{fn}$tel que $succ(e) = \{e'^+ \mid (snd(e), snd(e')) \in R\}$

6.2 Model checking CTL sur les fargments

Théorème 6.1 *Soit $M = (S, L, R, I)$ un fragment de structure de Kripke et soit la donnée d'une formule CTL. L'algorithme récursif suivant calcule l'ensemble des états $H(f) \subseteq S$ qui satisfont ou qui peuvent satisfaire f .*

- $H(p) = T^+(p) \cup U^+(p)$ tel que p est une proposition atomique

- $H(\neg f) = \{inT(s) \mid s \in (S - map(snd, H(f))\} \cup U^+(f)$
- $H(f \wedge g) = H(f) \sqcap H(g)$
- $H(f \vee g) = H(f) \sqcup H(g)$
- $H(AXf) = (\{InT(snd(e)) \mid \forall e \in S^+(f).succ(e) \subseteq T^+(f) \subseteq H(f)\} \cup \{inU(snd(e)) \mid \forall e \in S^+(f).succ(e) \subseteq (T^+(f) \cup U^+(f)) \ \ and \ \ \exists e' \in succ(e) \ \ such \ \ that \ \ e'^+(f)\}) \sqcup \{inU(s) \mid s \in border(M)\}$
- $H(EXf) = (\{InT(snd(e)) \mid \forall e \in S^+(f).\exists e' \in succ(e) \ and \ e'^+(f)\} \cup \{inU(snd(e)) \mid \forall e \in S^+(f).succ(e) \subseteq U^+(f) \subseteq H(f)\}) \sqcup \{inU(s) \mid s \in border(M)\}$
- $H(AGf) = \upsilon Z.(H(f) \sqcap AXZ)$
- $H(AFf) = \mu Z.(H(f) \sqcup AXZ)$
- $H(A(fUG)) = \mu Z.(H(g) \sqcup (H(f) \sqcap AXZ))$

Après l'application de l'algorithme récursif ci-dessus, nous avons $\forall s \notin H(f) \Rightarrow L(s, f) = false$. Les autres opérateurs, comme par exemple EG peuvent être déduits de l'ensemble des opérateurs cités ci-dessus, par exemple $H(EGf) = H(\neg(AF\neg f))$. Pour plus d'information le lecteur peut voir [MIT00].

Preuve. Pour la proposition atomique, nous pouvons voir que $H(p)$ est l'ensemble des états où la formule est vérifiée totalement ou partiellement.

Pour le cas $H(AXf)$, l'ensemble $(inT(snd(e)|\forall e \in S^+(f))$ représente l'ensemble des états où tous ses successeurs sont des états où la formule f est vérifiée. L'ensemble $\{inU(snd(e))|\forall e \in S^+(f).succ(e) \subseteq (T^+(f) \cup U^+(f)) \ et \ \exists \ e' \in succ(e) \ , tel \ que \ e' \in U^+(f)\})$, représente l'ensemble des états où leurs successeurs peuvent satisfaire f, par conséquent, cet ensemble est l'ensemble des états où $H(AXf)$ peut-être satisfaite. Par ailleurs, du fait qu'on connait pas les successeurs des états terminaux, on ajoute ces états au résultat du calcul $\{inU(s)|s \in border(M)\}$. En effet, l'algorithme pourra conclure dans la suite du calcul que ces états satisfont la formule $H(AXf)$.

Nous allons prouver la caractérisation du point fixe de l'opérateur AGf. La caractérisation du point fixe pour les autres opérateurs CTL ce fait de la même manière. L'ensemble 2^s de tous les sous ensembles de S forment un treillis sous l'ordre d'inclusion d'ensemble. Chaque élément S'du treillis peut également être considéré comme un prédicat sur S où le prédicat est considéré comme étant vrai ou incertain ($\perp$) exactement comme pour les états de S'. Le plus petit élément dans le treillis est l'ensemble vide, le prédicat qui lui est associé est la constante "faux". Le plus grand élément dans le treillis est l'ensemble S, pour lequel le prédicat peut être vrai. La fonction 2^s à 2^s sera appelée un transformateur de prédicat. Pour la démonstartion, nous suivrons une démarche similaire à [MIT00] ; d'abord nous pouvons voir que $\tau(Z) = H(f) \sqcap AXZ$ est monotone, D'autres parts $\sqcap$ est continue par

le théorème de *Tarski Knaster* nous pouvons conclure que AGf est le plus grand point fixe de $\tau(Z) = H(f) \sqcap AXZ$. ■

- **Proposition 6.2** *Le transformateur de prédicat $\tau(Z) = H(f) \sqcap AXZ$ est monotone.*

  Preuve. *Soit $P1 \subseteq P2$. Pour montrer que $\tau(P1) \subseteq \tau(P2)$. considérons un état arbitraire $s \in \tau(P1)$. Donc s satisfait ou peut satisfaire f. c'est à dire : $(L(s, f) = true$ ou $(L(s, f) = \perp)$ pour tous les états s'tel que $(s, s\prime) \in R$. Du fait que $P1 \subseteq P2$, $s\prime \in P2$ donc $s\prime \in \tau(P2)$.* ■

Exemple 6.2 *Prenons le fragment de la structure de Kripke M montré sur la Figure 1 6.6 dans le nœud1, la propriété à vérifier sur ce fragment est $AG(a \vee c)$:*

$H(a) = T^+(a) \cup U^+(a),\ \ T^+(a) = \{(1,2)\},$
$U^+(a) = \{(\perp,3),(\perp,4)\},\ H(a) = \{(1,2),(\perp,3),(\perp,4)\}$
$H(c) = \{(\perp,3),(\perp,4)\},\ H(a \vee c) = H(a) \sqcup H(c)$
$H(a \vee c) = \{(1,2)\} \cup \{(\perp,3) \vee (\perp,3),(\perp,4) \vee (\perp,4)\}\ H(a \vee c) = \{(1,2),(\perp,3),(\perp,4)\}$
$H(AG(a \vee c)) = \upsilon Z.(\{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ)$
$Z_0 = \{(1,1),(1,2),(1,3),(1,4)\}$
$AXZ_0 = \{(1,1),(1,2),(1,3),(1,4)\} \sqcup \{(\perp,3),(\perp,4)\} = \{(1,1),(1,2),(1,3),(1,4)\}$
$Z_1 = \{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ_0$
$Z_1 = \{(1,2),(\perp,3),(\perp,4)\}$
$AXZ_1 = \{(1,1),(\perp,2)\} \sqcup \{(\perp,3),(\perp,4)\} = \{(1,1),(\perp,2),(\perp,3),(\perp,4)\}$
$Z_2 = \{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ_1 = \{(\perp,2),(\perp,3),(\perp,4)\}$
$Z_3 = \{(\perp,2),(\perp,3),(\perp,4)\}$
$Z_3 = Z_2$

Le point fixe est atteint, l'algorithme arrête le calcul. A partir du théorème précédent nous concluons que : $L(1, AG(a \vee c)) = false$. Le résultat final de calcul sur le fragment dans le nœud 1 est donc $(0,1),(\perp,2),(\perp,3),(\perp,4)\}$

6.3 Model checking CTL Distribué

L'idée principale de l'algorithme de vérification distribuée consiste à prendre en compte le fait que la vérité d'une formule φ dans état s se trouvant sur un noeud donné peut dépendre de la vérité de cette formule sur des états qui peuvent appartenir à d'autres noeuds du réseau (voir la figure 6.2). A titre d'exemple, sur cette figure, en voulant vérifier la formule φ sur l'état

s', le calcul sur le nœud I commence avec $L(s', \varphi) = \perp$. On estime alors que cette formule peut être vérifiée en $s\prime$. Lorsque le nœud II termine le calcul, si la formule est désormais déclarée vérifiée en $s\prime$, le nœud numéro I recalcule les valeurs des formules sur ses états. Entre autre, dans le cas de $\varphi \in \{EGf, AGf, AFf, EFf, A(fUg), E(fUg)\}$. De même lorsque la formule est déclarée fausse après la fin du calcul dans le noeud II.

La principale différence entre le raisonnement de l'algorithme opérant sur les fragments vis à vis de l'algorithme distribué est que dans le cas de $\varphi \in \{EGf, AGf, AFf, EFf, A(fUg), E(fUg)\}$ nous considérons dans l'algorithme de fragments que f peut être vérifiée dans les états terminaux. Cependant dans la version distribuée nous considérons l'ensemble de la formule φ et non seulement une de ses parties. Le paramètre passé au transformateur de prédicats est alors de la forme :

- $AFp = \lambda Y.\mu Z.(p \vee Y \vee AXZ)$
- $EFp = \lambda Y.\mu Z.(p \vee Y \vee EXZ)$
- $AGp = \lambda Y.\upsilon Z.(p \vee Y \wedge AXZ)$
- $EGp = \lambda Y.\upsilon Z.(p \vee Y \wedge EXZ)$
- $A(p \cup q) = \lambda Y.\mu Z.(q \vee Y \vee (p \wedge AXZ))$
- $E(p \cup q) = \lambda Y.\mu Z.(q \vee Y \vee (p \wedge EXZ))$

Où $Y = \{s \in border(M) \mid L(s, \varphi) = True \ ou \ L(s, \varphi) = \perp\}$ et φ est une formule arbitraire représentée par l'un des six opérateurs décrits ci-dessus respectivement. Y représente ici l'absence d'information sur l'état terminal. Dans le cas où l'ensemble des états terminaux vérifiant la formule est déterminé, on conclut alors sur la valeur de vérité de la formule sur toute la structure de Kripke.

Lemme 6.1 *Le résultat de l'algorithme récursif ci-dessus ne peut être influencée que par la valeur de vérité de la formule à vérifier sur les états terminaux. De ce fait, seul le changement des valeurs de vérité des formules sur les états terminaux nécessite un autre calcul.*

Preuve.
On peut voir que l'algorithme de vérification ne fait intervenir que Y qui n'est autre que la valeur de vérité de la formule à vérifier sur les états terminaux. ■

Proposition 6.3 *La terminaison de l'algorithme distribué est atteinte lorsque il n'y a plus de changement des valeurs de vérité des formules des états terminaux.*

Preuve. À l'aide du lemme 6.1, nous pouvons voir que s'il n'y a pas de changement dans tous les états terminaux, aucun des noeuds ne relancera le calcul. De ce fait l'algorithme réparti arrive à un point fixe et se termine. ■

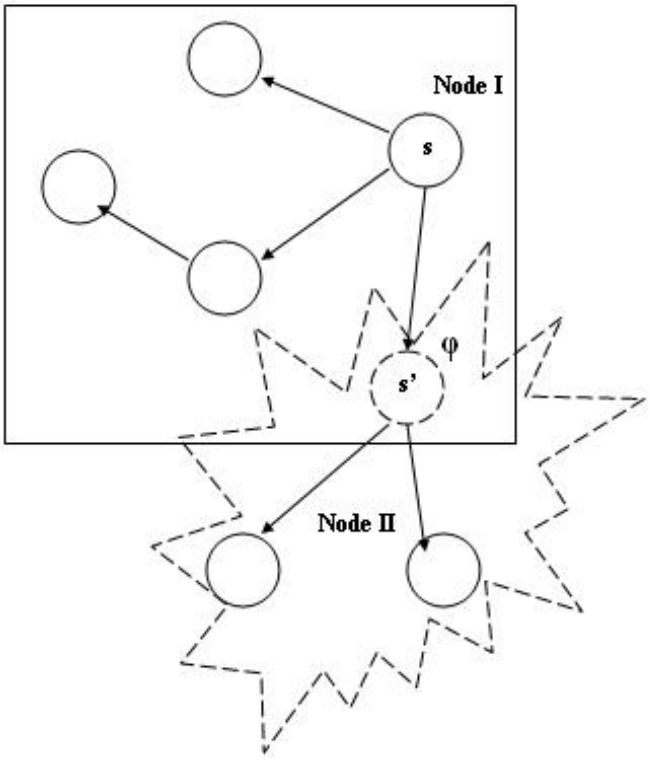


FIG. 6.2 – Model checking CTL distribué

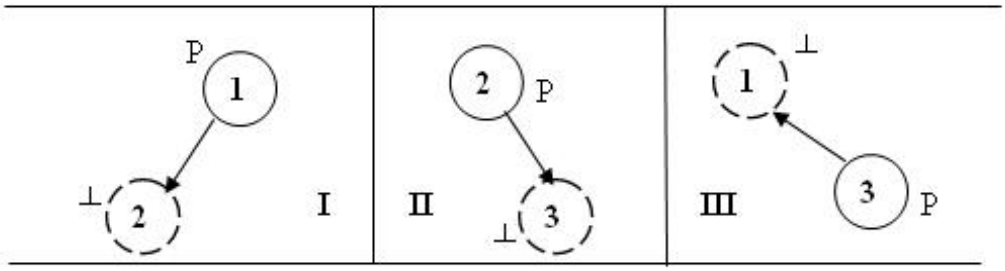


FIG. 6.3 – **Fragments de M distribués sur les noeuds**

Proposition 6.4 *Lorsque la terminaison est détectée et quelques valeurs de vérité demeurent indéfinies sur certains états alors ces états forment un circuit distribué sur différents noeuds.*

Preuve. Evident. ■

Proposition 6.5 *Lorsque la terminaison est détectée et quelques valeurs de vérité demeurent indéfinies sur certains états alors les valeurs de ces formules sont comme suit :*

1. $f = A(\phi_1 U \phi_2)$, $L(s, f) = false$
2. $f = AF\phi$, $L(s, f) = false$
3. $f = AG\phi$, $L(s, f) = true$

Preuve.

1. L'existence d'un cycle entre les noeuds sachant que la valeur de $f = A(\phi_1 U \phi_2)$ est indéfinie sur chacun des noeuds du cycle implique que la formule $\phi_1$est vraie sur tous les états s alors que aucun de ces états ne satisfait ϕ_2, de ce fait $L(s, f) = false$.
2. Remarquons que $f = AF\phi = A(true\ U\ \phi)$. Par l'utlisation du résultat précédent, on conclue que $L(s, f) = false$.
3. L'existence d'un cycle entre les noeuds sachant que la valeur de $f = AG\phi$ est indéfinie sur chacun des noeuds du cycle implique que la formule ϕ est vraie sur tout états s mais l'incertitude concerne les états terminaux, or le même scénarion se reproduit sur les différents noeuds, de ce fait on déduit que la formule est vraie sur tous les états du cycle, c'est à dire $L(s, f) = true$ (voir figure 6.3).

■

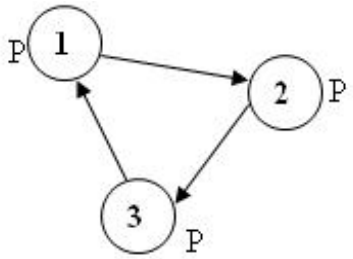


FIG. 6.4 – **Structure de Kripke avec cycle**

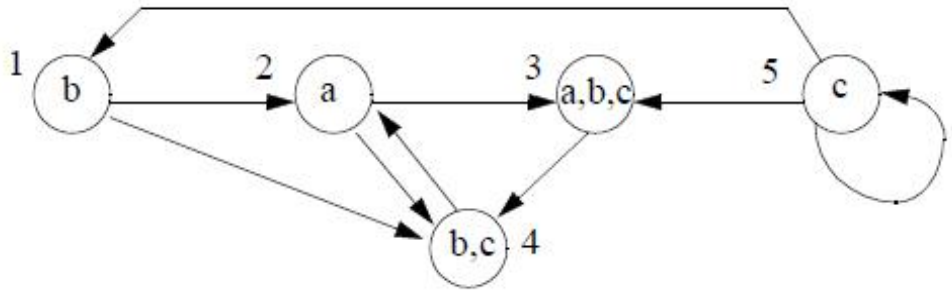


FIG. 6.5 – **Structure de Kripke**

La structure M montré à la figure 4, et la propriété à vérifier sur M est $AG(a \vee c)$:

$S = \{1, 2, 3, 4, 5\}, AP = \{a, b, c\}$,
$R = \{(1, 2), (2, 3), (3, 5), (5, 5),$
$(5, 1), (2, 4), (4, 2), (1, 4), (3, 4)\}, L(1) = \{b\}, L(2) = \{a\}, L(3) = \{a, b, c\}, L(4) = \{b, c\}, L(5) = \{c\}$
La distribution du système sur trois nœuds du réseau à l'aide de la fonction de partition h est montrée dans la figure 6.6.
$h : \{s_1, s_2, s_3, s_4, s_5\} \rightarrow \{node_1, node_2, node_3\}$
$h(1) = h(2) = 1, h(4) = 2, h(3) = h(5) = 3$

L'application de l'algorithme sur la structure M complète donne le même résultat que dans (2),$H(AG\ a \vee c) = \{2, 3, 4\}$, puisque toutes les informations nécessaires pour le calcul est disponible. Nous faisons la remarque que, dans le cas de l'application de l'algorithme sur la structure de Kripke complète, notre algorithme peut être simplifié à l'algorithme dans ([MIT00]), parce que $Y = \{\}$et les opérations $\sqcap$ et $\sqcup$ correspondent respectivement à $\cap$ et $\cup$.

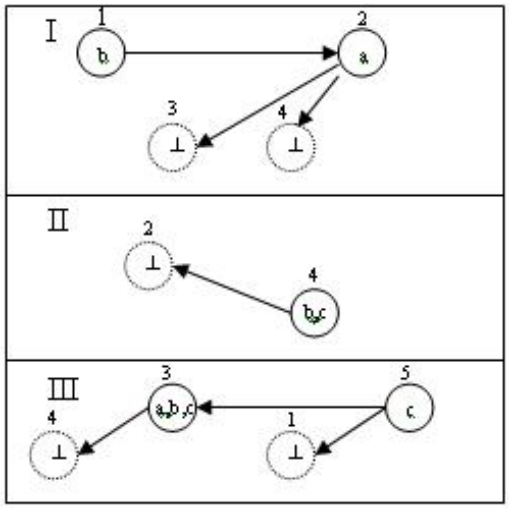


FIG. 6.6 – **Fragments de M distribués sur les noeuds**

***iteration 1* :**

Node I :

$H_p(a) = \{(1,2)\}$
$H_p(c) = \{\}$
$H_p(a \vee c) = \{(1,2)\}$
$Y = H_B(AG(a \vee c)) = \{(\perp,3),(\perp,4)\}$
$H(a \vee c) = Y \sqcup H_p(a \vee c) = \{(1,2),(\perp,3),(\perp,4)\}$
$H(AG(a \vee c)) = \upsilon Z.(\{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ)$
$Z_0 = \{(1,1),(1,2),(1,3),(1,4)\}$
$AXZ_0 = \{(1,1),(1,2),(1,3),(1,4)\} \sqcup \{(\perp,3),(\perp,4)\} = \{(1,1),(1,2),(1,3),(1,4)\}$
$Z_1 = \{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ_0$
$Z_1 = \{(1,2),(\perp,3),(\perp,4)\}$
$AXZ_1 = \{(1,1),(\perp,2)\} \sqcup \{(\perp,3),(\perp,4)\} = \{(1,1),(\perp,2),(\perp,3),(\perp,4)\}$
$Z_2 = \{(1,2),(\perp,3),(\perp,4)\} \sqcap AXZ_1 = \{(\perp,2),(\perp,3),(\perp,4)\}$
$Z_3 = \{(\perp,2),(\perp,3),(\perp,4)\}$
$Z_3 = Z_2$

Le point fixe est atteint, l'algorithme arrête le calcul et attend éventuellement de nouvelles informations relatives aux états terminaux. Nous concluons que $L(1, AG(a \vee c)) = false$ alors le résultat final du calcul de l'itération 1 dans le nœud 1 est : $\{(0,1),(\perp,2),(\perp,3),(\perp,4)\}$

Node II :

$H_p(a \vee c) = \{(1,4)\}$
$Y = H_B(AG(a \vee c)) = \{(\perp,2)\}$
$H(a \vee c) = Y \sqcup H_p(a \vee c) = \{(\perp,2),(1,4)\}$
$H(AG(a \vee c)) = \upsilon Z.(\{(\perp,2),(1,4)\} \sqcap AXZ)$
$Z_0 = \{(1,2),(1,4)\}$
$AXZ_0 = \{(\perp,2),(\perp,4)\}$
$Z_1 = \{(\perp,2),(\perp,4)\}$
$AXZ_1 = \{(\perp,2),(\perp,4)\}$
$Z_2 = \{(\perp,2),(\perp,4)\}$
$Z_3 = Z_2$

Node III :
$H_p(a \vee c) = \{(1,3),(1,5)\}$
$Y = H_B(AG(a \vee c)) = \{(\perp,1),(\perp,4)\}$
$H(a \vee c) = Y \sqcup H_p(a \vee c) = \{(1,3),(1,5),(\perp,1),(\perp,4)\}$
$H(AG(a \vee c)) = \upsilon Z.(\{(1,3),(1,5),(\perp,1),(\perp,4)\} \sqcap AXZ)$
$Z_0 = \{(1,1),(1,3),(1,4),(1,5)\}$
$AXZ_0 = \{(1,1),(1,3),(1,4),(1,5)\}$
$Z_1 = \{(\perp,1),(1,3),(\perp,4),(\perp,5)\}$
$AXZ_1 = \{(\perp,3),(\perp,5)\} \sqcup \{(\perp,1),(\perp,4)\}$
$AXZ_1 = \{(\perp,1),(\perp,3),(\perp,4),(\perp,5)\}$
$Z_2 = \{(\perp,1),(\perp,3),(\perp,4),(\perp,5)\}$
$Z_3 = \{(\perp,1),(\perp,3),(\perp,4),(\perp,5)\}$
$Z_3 = Z_2$
Point fix atteint. .

***iteration 2* :**

Node III :

Nous pouvons faire un autre calcul seulement sur le nœud III (Lemme 6.1). Seulement dans nœud 3 nous avons un changement dans les états terminaux, parce que la valeur de vérité dans l'état 1 est modifiée :
$H(a \vee c) = \{(1,3),(1,5),(\perp,4)\}$
$Z_0 = \{(1,1),(1,3),(1,4),(1,5)\}$
$AXZ_0 = \{(1,1),(1,3),(1,4),(1,5)\}$
$Z_1 = \{(1,3),(\perp,4),(1,5)\}$
$AXZ_1 = \{(\perp,3),(\perp,4),(\perp,1)\}$

$Z_2 = \{(\perp, 3), (\perp, 4)\}$
$Z_3 = Z_2 = \{(\perp, 3), (\perp, 4)\}$
De l'algorithme 6.3, nous concluons que $L(5, AG(a \vee c)) = false$ alors le résultat final dans le nœud III est :
$\{(\perp, 3), (\perp, 4), (0, 5)\}$

En utilisant la proposition 6.3 , du fait qu'aucun changement ne s'est produit dans les états terminaux, le calcul termine et l'algorithme distribué s'arrête dans l'itération numéro 2. En utilisant la proposition6.5, nous concluons que $L(2, AG(a \vee c)) = true$,$L(3, AG(a \vee c)) = true$, $L(4, AG(a \vee c)) = true$.

Le résultat final de l'ensemble du calcul sur les trois nœuds est :$\{(1, 2), (1, 3), (1, 4)\}$.

6.4 Conclusion

Dans ce chapitre nous avons présenté un algorithme distribué de vérification qui exploite l'information au niveau des états du graphe de maximalité[ZEAB10]. Cet algorithme donne le résultat de la vérification malgré l'insufisance d'information résultante de la distribution du graphe d'état sur plusieurs nœuds. Ceci est assuré par l'échange d'information manquante entre les nœuds et qui concerne des états terminaux. Un travail similaire au notre est celui de [LK]. Ce dernier a traité le même problème en utilisant la notion de suppositions avec une approche impérative, ce qui complique les preuves, la compréhension et la réutilisation de leur approche dans l'industrie. Notre approche peut être considérée comme la meilleur sur le plan de la formalisation car elle utilise la notion de la logique à trois valeurs de Kleen. Par ailleurs la présentation est faite en utilisant une approche fonctionnelle, ce qui facilite les preuves de la validité des algorithmes. De plus, la réutilisation de notre algorithme dans l'industrie est facile grâce à l'aspect modulaire des fonctions. Notons aussi que le travail de [LK] est basé sur le modèle de Kripke, cela ne lui permet pas de verifier certaines propriétés dont la preuve est possible sur le modèle de maximalité, tel que l'autoconcurrence. Un autre point à souligner dans le travail de [LK] est qu'il ne détaille pas la manière dont les fragments sont traités lors de la vérification.

Chapitre 7

Implémentation et résultats

Dans le présent travail nous avons enrichi l'environnement de vérification formelle FOCOVE par deux outils « générateur distribué » et «évaluateur distribué ». Dans ce chapitre, nous présentons la conception globale et les techniques adoptées pour implémenter ces outils. L'outil «générateur distribué » permet de traduire une spécification LOTOS en système de transitions étiquetées maximales alpha réduit distribué sur plusieurs machines sous forme de fragments, ce qui permet sa vérification et sa validation. Cette génération réduite et distribuée contribue à la maîtrise du problème de l'explosion combinatoire du graphe d'états. En effet l'algorithme proposé effectue la réduction à la volet du STEM. L'outil «évaluateur distribué » à pour objectif de vérifier les propriétés écrites en logique temporelle CTL sur les fragments distribués. L'outil « générateur distribué » à comme entrée une spécification LOTOS écrite sous une forme intermédiaire à partir de laquelle les fragments du STEM distribué sont générés. Ces fragments sont stockés dans des fichiers sur chaque nœud pour pouvoir ensuite être restaurés. Ils peuvent être visualisés graphiquement en utilisant l'outil STEM Viewer ou l'outil Dotty.

Notons que le langage de programmation Erlang est choisi pour l'implémentation des noyaux de ces outils.

7.1 L'environnement FOCOVE :

FOCOVE est une boîte à outils pour la compilation, la vérification et la validation de spécifications écrites en LOTOS. La conception de l'architecture globale a été élaborée par le Professeur Djamel-Eddine Saidouni laboratoire MISC du département d'informatique de l'université de constantine en l'année 2000.

FOCOVE intègre un ensemble cohérent d'outils permettant de traiter des spécifications comportementales et logiques, en utilisant les méthodes basées sur les modèles. L'environnement comprend divers outils, nous pouvons les classer comme suit :

- **Les compilateurs :** utilisés en amont, leur rôle est de traduire le programme (spécification formelle) à vérifier vers un modèle sémantique de parallélisme. FOCOVE contient cinq compilateurs :
  - ICC (Interleaving CCS Compiler) : En entrée, il prend le programme CCS à vérifier et il fournit comme sortie un STE.
  - ILC (Interleaving LOTOS Compiler) : Il permet d'obtenir un STE à partir d'une spécification LOTOS.
  - LOTOSTEM : Le modèle sémantique considéré par cet outil est celui des arbres maximaux, ces arbres sont dérivés à partir de spécifications LOTOS.
  - LOTOSTEMv2.0 : Génère à la volet un STEM alpha réduit.
  - LOTOSGPM : C'est un outil d'ordre partiel, il fournit à la volet un graphe qui couvre le STEM initial modulo l'équivalence de traces de Mazurckiewicz.
- **Les vérificateurs :** Utilisés en aval, ils effectuent des vérifications sur les graphes générés par les compilateurs. FOCOVE couvre les quatre types de vérifications :
  - La vérification comportementale : deux outils permettent respectivement la réduction et la comparaison des STEs par rapport à des relations d'équivalence comportementales à savoir la bissimulation forte et faible.
  - La vérification logique : LotoStem contient un model-checker qui permet de vérifier des propriétés exprimées en logique temporelle arborescente (CTL) sur le STEM généré.
  - La vérification symbolique : MSMC (Maximality based Symbolic Model Checking) : Cet outil prend comme argument une spécification de système à vérifier sous forme d'un système de transitions étiquetées maximales, et une spécification d'une propriété à vérifier exprimées en logique temporelle arborescente CTL. L'implémentation de cet outil est basée sur l'utilisation des diagrammes de décision binaire (BDDs) qui permettent d'éviter la construction explicite du graphe d'états.
  - La vérification par test : C'est un outil de mise en œuvre de l'approche test, apportant une certitude mathématique par génération de tests de conformité sous forme d'une structure appelée : graphe de refus, et par extension et adaptation du calcul de bissimulation pour la vérification des relations de conformité d'une implémentation par rapport à sa spécification.
  - La boîte à outils FoCove fournit aussi deux outils permettant la visualisation graphique du STE (STEViewer) et du STEM (STEMViewer).

7.2 Choix du langage d'implémentation

En suivant une démarche classique, la première étape pour la mise en œuvre de nos outils consiste à choisir le langage de programmation. A cette fin nous avons choisi dans la première étape de génération distribuée le langage Haskell pour plusieurs raisons. L'une des raisons principales réside dans l'utilisation intensive des structures de listes. Entre autre, la liste des états, la liste des transitions etc. D'ou le choix des langages fonctionnels. L'autre argument revient au fait que les langages fonctionnelles possèdent la propriété de *transparence référentielle,* ce qui rend la preuve de la sûreté des programmes facile à établir. La troisième raison découle du fait que les langages fonctionnels sont très appropriés pour la mise en œuvre des règles sémantiques opérationnelles. En effet, grâce à la propriété du *Pattern Matching,* il est facile d'établir la mise en œuvre de la sémantique opérationnelle de Maximalité. Par conséquence nous avons défini un langage intermédiaire qui nous a permi d'exploiter la propriété du Pattern Matching pour implémenter La sémantique opérationnelle structurelle de maximalité en utilisant les types de données de Haskell par le biais du constructeur *data* comme suit :

```
> type Action = String
> type Appel_effectif = [Action]
> type Parametres_formels = [Action]
> type Id_processus = Int
>
> data Expr = Stop
> | Exit
> | Fix
> | Prefixe Action Expr
> | Hide [Action] Expr
> | Choix Expr Expr
> | Sequence Expr Expr
> | Parallel Expr Expr [Action]
> | Interrupt Expr Expr
> | RecX Expr
> | Appel Appel_effectif Parametres_formels Expr deriving (Show,Eq)
>
>
>
> type Processus = (Id_processus , Parametres_formels, Expr)
> type System = [Processus]
>
>
> type Action_max = (Action,Int)
```

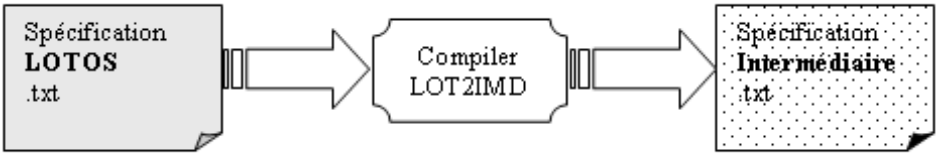
```
>
>
> data Config = StopConf [Action_max ]
> |ExitConf [Action_max ]
> |FixConf [Action_max ]
> |PrefixeConf [Action_max ] Action Expr
> |HideConf [Action] Config
> |ChoixConf Config Config
> |ParallelConf Config Config [Action]
> |SequenceConf Config Config
> |InterruptConf Config Config
> |AppelConf [Action_max ] Appel_effectif Parametres_formels Expr
> |RecXConf Config deriving (Show,Eq)
>
> type Etat = Int
> type Atom = ([Action_max ],Action, Action_max )
> type Elem_Config = (Config,Etat)
> type List_Config = [Config]
> type Graph = [(Config,Atom,Config)]
```

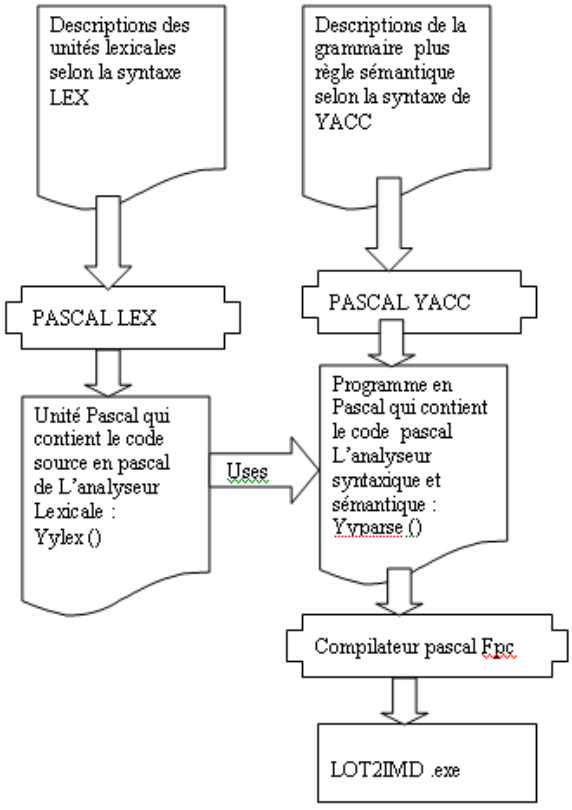
Tel qu'on peut le voir, Haskell est un langage typé statiquement : La vérification de la compatibilité entre les types des paramètres formels et des paramètres d'appel est effectuée au moment de la compilation, ce qui fait accroître l'efficacité et la sûreté de l'exécution. Il est clair que le constructeur data n'est autre qu'une description de la syntaxe de notre langage intermédiaire par une Bakus-nor-Formalism. Dans le context de la théorie des types, ce dernier représente un type récursif.

7.3 Le compilateur LOT2IMD

Dans une deuxième étape nous avons développé un compilateur qui transforme une spécification écrite en basic LOTOS vers le langage intermédiaire selon le schéma suivant :

La mise en œuvre de ce compilateur est basée sur l'approche de compilation ascendante LR en utilisant les outils LEX et YACC dédiés à la génération automatique de compilateur. Les outils LEX et YACC standard génèrent le code source du compilateur en langage C. Il existe plusieurs versions de LEX et YACC selon le langage de programmation utilisé. Par exemple, dans le cas du langage fonctionnel Ocaml, il existe Ocamllex et Ocamlyacc. Pour la réalisation du compilateur LOT2IMD, nous avons utilisé Pascal Lex et Pascal Yacc. Le schéma suivant explique le processus de compilation :





A titre d'exemple, pour la spécification LOTOS suivante :
$P[a,b] = (a; stop[]b; stop)|||(a; stop[]b; stop)$

Aprés le processus de compilation, l'outil LOT2IMD génère la spécification intermédiaire suivante :

$ParallelConf(ChoixConf(PrefixeConf[]"a"Stop)(PrefixeConf[]"b"Stop))$
$(ChoixConf(PrefixeConf[]"a"Stop)(PrefixeConf[]"b"\ \ Stop))[].$

L'approche de compilation utilisée pour obtenir le code intermédiaire est l'approche d'analyse ascendante basée sur une définition dirigée par la syntaxe, c'est à dire une association entre les règles de productions et les règles sémantiques.

7.4 Implémentation des règles sémantiques de Maximalité

Pour l'implémentation des règles sémantiques de maximalité il est clair de voir que la relation de transition est définie sur les configurations, ce qui nous mènera à faire le pattern matching sur ces dernières. Ainsi nous faisons la remarque que les langages fonctionnels permettent de définir des fonctions d'ordre supérieur (higher order function), c'est à dire manipuler les fonctions en tant que paramètres d'autres fonctions ou être retournées comme résultats d'un appel de fonctions. Ceci facilitera l'implémentation des règles sémantiques. Cette propriété est aussi utile dans la définition de la sémantique dénotationnelle. Les règles sémantiques sont semblables aux règles d'inférence constituées de prémisse et de conclusion. D'où l'utilisation de l'instruction fonctionnelle "Let" qui n'est autre que la beta réduction du lambda calcul. A titre d'exemple, l'implémentation de l'opérateur de choix de LOTOS est comme suit :

$> rule\ n\ (ChoixConf xy) = let(xs, ys) = ((rule(getM(x))x), (rule(getMy)y))in(fstxs++fstys, sndxs++sndys)$

Il est Clair l'exécution de cette règle fait le pattern Matching sur ChoixConf, et comme cette règle possède des prémisses et une conclusion, l'instruction "Let" calcule les prémisses afin d'obtenir la conclusion. L'inférence de type de cette fonction est :

$rule :: Int -> Config -> (List\_Config, Graph)$.

Cette fonction est donc une fonction d'ordre supérieur. La fonction "rule" à comme paramètre un entier et retourne comme résultat une fonction. Cette dernière prend comme paramètre une configuration et retourne comme résultat un couple qui contient la liste des nouvelles configurations comme première composante et la liste des transitions comme deuxième composante. Il est clair de voir qu'à partir de cette fonction on obtient une liste énumérative des transitions.

7.5 Transformateur

Afin de réutiliser les éditeurs graphiques existants tel que Dotty, nous avons développé un composant assurant la transformation vers le format spécifique de cet éditeur. Dotty est un outil qui utilise l'analyseur "dot", qui possède des algorithmes de placement permettant d'arranger les éléments du graphe. L'intérêt de cet outil réside dans sa facilité d'utilisation, ses performances et sa capacité d'afficher des graphes complexes. Cet outil est disponible Gratuitement.

7.5.1 Dotty

L'outil Dotty permet de visualiser des graphes, en chargeant un fichier textuel dont la syntaxe est assez simple (fichier d'extension .dot), voir l'exemple 7.1 . Le langage utilisé pour construire ces fichiers textuels est basé sur trois sortes d'éléments : les graphes, les nœuds et les arcs.

Chaque nœud est défini par un nom spécifique et unique pour un graphe donné. Un nœud est créé dès que son nom apparaît dans le fichier pour la première fois. L'arc est créé lorsque deux noms de nœud sont reliés par l'opérateur « -> ». Le graphe principal (mot clé "*digraph*" dans l'exemple ci dessous), est l'élément qui contient les nœuds, les arcs et les sous graphes. Un sous graphe est un graphe encadré par un rectangle. Il contient ses propres nœuds et arcs, et n'est dessiné que lorsque son nom est préfixé par le mot "*cluster*".

Remarquons qu'un sous graphe n'est affiché que lorsqu'il contient des nœuds, un sous graphe vide ne peut donc pas être affiché.

Exemple 7.1

```
digraph G{size="=1"; taille [shape=box];
taille [label="4 Transitions et 4 \U{e9}tats"]
edge [color = green,fontname = verdana, fontsize = 8, fontcolor = black];
node [fontname = verdana, fontsize = 12, fontcolor = green]
0->1[label="p"];
1[label="{p}"];
1->2[label="c"];
2[label="{c}"];
2->3[label="c"];
3[label="p"];
3->1[label="p"];
1[label="p"];
}
```

Remarque 7.1 *L'outil Dotty a été développé par AT&T Labs, au sein du projet GraphViz (http ://www.research-att.com/sw/tools/graphviz/), c'est un ensemble de logiciels libres qui permettent le calcul et l'affichage de graphes.*

7.6 Résultats expérimentaux

Pour évaluer la faisabilité de notre approche, nous avons implémenté l'algorithme 5.1. La mise en œuvre a été faite en Haskell sur un réseau de dix machines Pentium de 2 GHz de

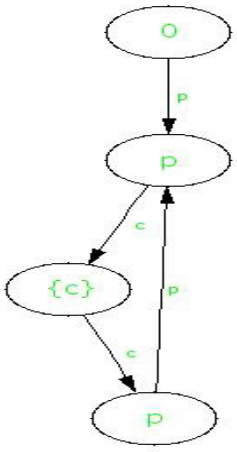


FIG. 7.1 – Image resultante de la compilation de l'exemple 7.1 par l'outils dot

fréquence d'horloges avec 128 Mo de RAM liées par un réseau Ethernet à 100 Mbits/s. Les primitives de communication sont construites au-dessus de TCP / IP en utilisant les sockets. L'objectif principal était de montrer que, la version distribuée de construction de l'espace d'états des stems ne devrait pas exclure l'usage des techniques de réduction centralisée liées aux stems, en particulier la réduction modulo l'alpha équivalence. L'exemple que nous avons considéré a mit en évidence le taux de réduction, ce dernier devient considérable lorsque le degré de parallélisme augmente. A titre d'exemple, pour six processus parallèles décrits par l'expression LOTOS :

$System\ Test[a,b,c,d,e,f,g] := P1[a]|||P1[b]|||P1[c]|||P1[d]|||P1[e]|||P1[f]|||P1[g]$
where
$Process\ P1[a] := a; P1[a]$ *endproc*
Endsys

L'application de l'algorithme 5.1 sans la prise en compte de la réduction modulo l'alpha équivalence génère 1957 états dispersés sur les nœuds selon la figure 7.2, alors que la considération de la réduction modulo l'alpha équivalence pour ce même algorithme n'en produit que 64 états dispersés selon la figure 7.3. On remarque que le taux réduction varie entre 88 et 99% (voir figure 7.4).

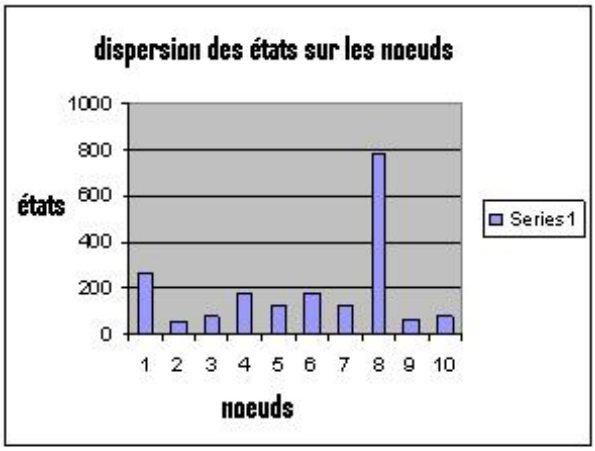


FIG. 7.2 – Dispersion des états sur les noeuds sans réduction

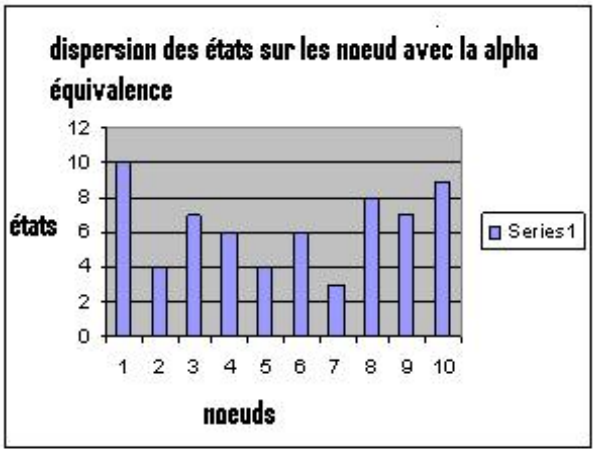


FIG. 7.3 – Distribution des états sur les noeuds avec la alpha équivalence

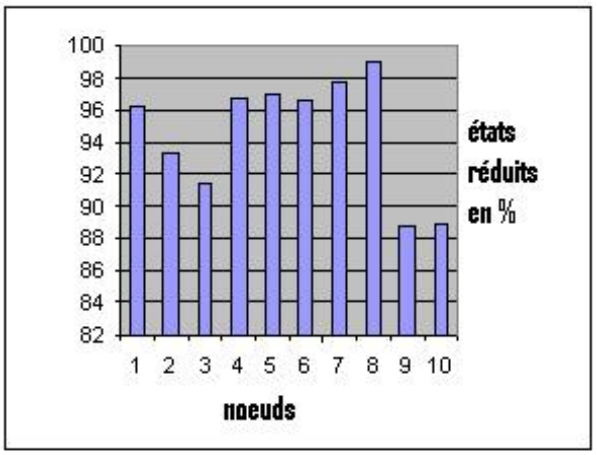


FIG. 7.4 – Taux de réduction en % sur chaque noeud

Chapitre 8

Conclusion générale et perspectives

Ce travail se situe dans le cadre de la vérification formelle des systèmes concurrents. Cette vérification est basée sur la construction d'un modèle du système à vérifier. Ce modèle est habituellement donné par un graphe d'états. La taille importante de ce modèle est un facteur limitatif majeur de l'applicabilité de ce type de vérification. Ce problème est connu sous le nom de l'explosion combinatoire du graphe d'états.

L'objectif de ce travail consistait en l'étude et la mise en œuvre d'une méthode de génération et vérification distribuées d'une spécification LOTOS basée sur la sémantique de maximalité, permettant ainsi de remédier au problème de l'explosion combinatoire du graphe d'états d'une part et de considérer la non atomicité temporelle et structurelle des actions d'autre part. Pour cela, nous avons exposé différents aspects dans notre étude.

D'abord, nous avons présenté au niveau du chapitre 2, la sémantique de maximalité de Basic LOTOS ainsi que le modèle des STEMs, utilisé comme modèle logique de vérification, ce modèle permet de conserver la nature parallèle des systèmes concurrents. Le chapitre 3 a été consacré à la méthode de vérification basée modèle (Model checking), il a mit en évidence les algorithmes utilisés pour assurer la vérification sur la structure des STEMs. Par ailleurs il mit la lumière sur l'avantage de la propriété de l'autoconcurrence. Afin d'enrichir notre exposé, nous avons présenté dans le chapitre 4 la majorité des approches de génération et de vérifications parallèles est distribuées mentionnées dans la littérature. Dans le chapitre 5 nous avons proposé notre méthode de génération distribuée des STEMs avec réduction préservant la alpha équivalence, le résultat de cette génération est l'ensemble du STEM répartie en fragments sur les nœud participant dans l'opération de génération. Dans le chapitre 6 nous avons présenté notre algorithme de vérification distribuée opérant sur les fragments, cet algorithme utilise la logique à trois valeurs dites de Kleen. Enfin, Dans le chapitre 7, nous avons présenté l'implémentation de la méthode proposée dans les chapitres 5 et 6 ainsi que quelques résultats expérimentaux en utilisant l'outil développé. L'importance de la réduction est mise en évidence à travers la considération d'un exemple sur lequel nous avons appliqué notre algorithme. Le taux de réduction avoisine les 99% sur certains nœuds. Nous faisons remarquer que l'importance de ce taux est fonction du degré de parallélisme existant dans

l'application. Un des points méritant quelques commentaires concerne le choix de la fonction de partition qui évite la communication potentielle entre les nœuds. Ce problème peut être réduit par l'utilisation des propriétés structurelles de LOTOS.

Par ailleurs, ce travail peut se poursuivre dans plusieurs directions. Il serait intéressant de développer des algorithmes de réduction de stems modulo d'autres relations d'équivalence tels que les relations de test de conformité, de bissimulation de maximalité etc. Nous pensons également que l'utilisation de langages de programmation plus appropriés pour la distribution, entre autre le langage Erlang, pourra augmenter les performances de l'algorithme sur une plateforme contenant un nombre important de nœuds. En plus, étudier la complexité des algorithmes proposés au but d'améliorer l'efficacité de l'outil développé en temps de réponse et en espace mémoire et l'expérimentation de l'outil sur des spécifications d'applications industrielles dans le but d'évaluer le gain obtenu en appliquant cette méthode par rapport aux autres méthodes.

Bibliographie

[Aho] Ulman Aho. Compilers principals techniques and tools.

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[Bar02] J. Barnat. Using verified property to partition the state space in LTL modelchecking. *Proceedings of the Summer School on Modelling and Verifying Parallel Processes*, 2002.

[BBL+99] B. Bernard, M. Bidoit, F. Laroussine, A. Petit, and Ph. Schnoebelen. *Vérification de logiciels: Techniques et outils du Model-Checking* . Paris, vuibert edition, 1999.

[BC03a] L. Brim J. Barnat and J. Chaloupka. Parallel breadth-first search LTL model checking. *18th IEEE International Conference on Automated Software Engineering*, pages 106–115, 2003.

[BC03b] L. Brim J. Barnat and J. Chaloupka. Prallel breadth-first search LTL model checking. *In 18th IEEE International Conference on Automated Software Engineering*, pages 106–115, 2003.

[BCM+92a] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCM+92b] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BH03] I. Cernţa L. Brim and L. Hejtmanek. Parallel algorithms for detection of NegativeCycles. 2003.

[BJ03] L. Brim J. Barnat and J.Chaloupka. Distributed memory LTL model checking based on breadth first search. *October*, 2003.

[Bou05a] Mustapha Bourahla. Distributed CTL model checking. *IEE software engineering*, 152(6), 2005.

[Bou05b] Mustapha Bourahla. Partitioning state spaces of cocurrent transition systems. *International Arab Journal of Information Technology*, 2, 2005.

[Bry86] R. E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computer Sciences*, 37:77–121, 1986.

[BS01] L. Brim J. Barnat and J. Stobrna. Distributed LTL model-checking in SPIN. *Proc. SPIN Workshop on Model Checking of Software*, 2057 of LNCS:200–216, 2001.

[BS05] N. Belala and D. E. Saïdouni. Non atomicité dans les modèles temporisés. Research Report IPG05-001, Laboratoire LIRE, Université Mentouri, 25000 Constantine, Algérie, February 2005.

[CE81] E. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *IBM Workshop on logics of programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CES86] 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*, 8(2):244–263, April 1986.

[CG99] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical programming*, Springer-Verlag :85:277–311, 1999.

[Cou] Jean-Michel Couvreur. Contribution ţa l'algorithmique de la vérification,2004.

[CS95] J. P. Courtiat and D. E. Saïdouni. Relating maximality-based semantics to action refinement in process algebras. In D. Hogrefe and S. Leue, editors, *IFIP TC6/WG6.1, 7th Int. Conf. on Formal Description Techniques (FORTE'94)*, pages 293–308. Chapman & Hall, 1995.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science : Formal Models and Semantics, Ch. 16*, volume B, pages 995–1072. Elsevier, 1990.

[FDI94] FDIV replacement program. *Intel White Paper*, November 1994.

[F.L99] R.Sisto F.Lerda. Distributed-memory model-checking with SPIN. *Proc.5th Int. SPIN Workshop*, 1999.

[Gle96] J. Gleick. A bug and a crash - sometimes a bug is more than a nuisance. *New York Times Magazine*, December 1996.

[GS00] T. Heyman D. Geist O. Grumberg and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. 12th Int. Conf. On Computer Aided Verication LNCS 1855*, Springer 2000.

[HG80] H.Gazit and G.L.Miller. An improved parallel algorithm that computes the BFS numbring of directed graph. *information processing Letters*, 28(2):61–65, 1980.

[Hol03] G. Holzmann. The SPIN model-checker. 2003.

[HV00] G. Behrmann T. S. Hune and F. W. Vaandrager. Distributed timed model-checking - how the search order matters. In *12th Int.Conf.Computer Aided Verication LNCS 1855*, pages 216–231, Springer 2000.

[IB05] Cornelia P. Inggs and Howard Barringer. CTL* model checking on a shared-memory architecture. *Department of Computer Science*, 2005.

[I.S01] H.Garavel R.Mateescu I.Smarandache. Parallel state space construction for model-check. *LNCS 2057*, 2001.

[K.Y02] L.Brim J.Crhova K.Yorav. Using assumptions to distribute CTL model checking. parallel and distributed model checking PDMC'2002. *Electronic Notes in Theoretical Computer Science 68(4)*, 2002.

[LA00] B.A.Hendrickson L.Fleischer and A.Pinar. On identifying strongly connected components in parallel in solving irregularly structred problems in parallel. *Lecture Notes in computer Science*, SpringerVerlag 1800:505–512, 2000.

[Lam83] L. Lamport. What good is temporal logic ? . *Information processing letters*, 83:657–668, 1983.

[LK] J ;Crhova R. Haverkot L.Brim and K.Yorav. Using assumption to distribute CTL model checking. mazaryk university czech republic 2002.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking in SPIN. *The SPIN Workshop*, 1680 of Lecture Notes in Computer Science, 1999.

[LT93] N. G. Leveson and C. S. Turner. Investigation of the therac-25 accidents. *IEEE Computer 26(7)*, pages 18–41, 1993.

[McM92] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[MD5] Http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/md5/.

[Mic] Diaz Michel. Les réseaux de petri modèles fondametaux ,2001.

[MIT00] Model checking. *MIT*, 2000.

[O.G98] K.Laster O.Grumberg. Modular model-checking of software. *Proc. TACAS'98 LNCS 1384*, pages 20–35, 1998.

[PLBJ04a] I. Cernà P.Moravec L. Brim and J.Simasa. Accepting predecessor are better thenBack eges in distributed LTL model checking. October 2004.

[PLBJ04b] I. Cernţa P.Moravec L. Brim and J.Simasa. Accepting predecessor are better then back eges in distributed LTL model checking. *October*, 2004.

[RAH03] Messaoud RAHIM. Vérification distribuée des systèmes de grandes tailles. Master's thesis, Institut National d'Informatique, 2003.

[Saï96] D. E. Saïdouni. *Sémantique de maximalité: Application au raffinement d'actions en LOTOS*. PhD thesis, LAAS-CNRS, 7 av. du Colonel Roche, 31077 Toulouse Cedex France, 1996.

[SB03a] D. E. Saïdouni and N. Belala. Vérification de propriétes exprimées en ctl sur le modèle des systèmes de transitions étiquetées maximales. In *CIP'2003, Conférence Internationale de Productique*, Alger, Algérie, octobre 2003. CDTA.

[SB03b] D. E. Saïdouni and N. Belala. Vérification de propriétes exprimées en ctl sur le modèle des systèmes de transitions étiquetées maximales. Research report, Laboratoire LIRE, Université Mentouri, 25000 Constantine, Algérie, 2003.

[SB04] D. E. Saïdouni and N. Belala. Straightforward adaptation of interleaving-based solutions for true concurrency-based logic verification approaches. In *Proceedings of International Conference on Complex Systems (CISC'2004)*. University of Jijel, Algeria, September 6-8th 2004.

[SB05] D. E. Saïdouni and N. Belala. Using maximality-based labeled transition system model for concurrency logic verification. *The International Arab Journal of Information Technology (IAJIT)*, 2(3):199–205, July 2005. ISSN: 1683-3198.

[SC94] D. E. Saïdouni and J. P. Courtiat. Syntactic action refinement in presence of multiway synchronization. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Workshop on Semantics of Specification Languages (SoSL'93)*, Workshops in Computing, pages 289–303, Utrecht, 1994. Springer-Verlag.

[SC96] D. E. Saïdouni and J.P. Courtiat. Une comparaison des sémantiques de maximalité et de causalité de basic lotos. In *CFIP : Ingenierie des protocols, ENSIAS, Rabat, Maroc*, 1996.

[SC03] D. E. Saïdouni and J. P. Courtiat. Prise en compte des durées d'action dans les algèbres de processus par l'utilisation de la sémantique de maximalité (version étendue). Technical Report 03243, LAAS-CNRS, 7 avenue du colonel Roche, 31077, Toulouse Cedex France, may 2003.

[Sch] P. Schnoebelen. Vérification de logiciels , techniques et outils du model-checking. *Masson edition, 1999.*

[SD97] U. Stern and D. Dill. Parallelizing the mur verifier. *Proc. 9th Int Conf on Computer Aided Verication LNCS 1254.*, Springer 1997.

[Tar72] R. E. Tarjan. Depth-frst search and linear algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[ZEAB08] Djamel Eddine Saidouni Zine El Abidine Bouneb. Parallel MLTS construction preserving the alpha equivalence reduction. *IRECOS International review on computers and software*, 2008.

[ZEAB09] Djamel Eddine Saidouni Zine El Abidine Bouneb. Parallel state space construction for a model CheckingBased on maximality semantics. In *Conference CISA*. Americain institute of physics AIP ,NASA, 2009.

[ZEAB10] Djamel Eddine Saidouni Zine El Abidine Bouneb. Distributed maximality based CTL model checking. *IJCSI International Journal of Computer Science*, Vol 7, Issue 3, 2010.